THE INTEGRATION OF DIFFERENT FUNCTIONAL AND STRUCTURAL PLANT MODELS

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
Qinqin Long

from Chongqing, China,
Göttingen 2019

Thesis Committee

Prof. Dr. Winfried Kurth

(Department Ecoinformatics, Biometrics and Forest Growth, University of Göttingen)

Prof. Dr. Kerstin Wiegand

(Department of Ecosystem Modelling, University of Göttingen)

Members of Examination Board

Prof. Dr. Winfried Kurth

(Department Ecoinformatics, Biometrics and Forest Growth, University of Göttingen)

Prof. Dr. Jochem Evers

(Department of Plant Sciences, Wageningen University & Research)

Further members of the Examination Board

Prof. Dr. Kerstin Wiegand

(Department of Ecosystem Modelling, University of Göttingen)

Prof. Dr. Dieter Hogrefe

(Telematics group, University of Göttingen)

Prof. Dr. Marcus Baum

(Data Fusion group, University of Göttingen)

Prof. Dr. Jens Grabowski

(Software Engineering for Distributed Systems group, University of Göttingen)

Date of the oral examination: May 20, 2019

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.
- Dr. Christophe Pradal - for his insightful discussions, support, comments and indispensable cooperation.
- Dr. Christophe Pradal, Dr. Reinhard Hemmerling and Uwe Mannl - for their contribution of a prototype of the interface between GroIMP and OpenAlea
- Dr. Michael Henke, Dr. Ole Kniemeyer - for their basic work, essential support, comments and precious friendship.
- My father Dehua Long, mother Qiongyuan Song, - for their infinite love and support.
- My wife Zhihua Gong, - for her infinite love, companionship, care and support.
- Dr. Evelyne Costes - for her motivating support, insightful comments, indispensable cooperation, and warm host of my research stays and project workshop.
- Prof. Dr. Gerhard Buck-Sorlin, Prof. Dr. Paul-Henry Cournede - for their motivating support, insightful comments, indispensable cooperation, and the warm host of the project workshops.
- Dr. Vincent Migault, Dr. Benoît Pallas, Dr. Benoît Bayol - for their helpful support and comments, and indispensable cooperation.
- Dr. Faustino Hilario Chi, Dr. Johannes Merklein, Mr. Aleksi Tavkhelidze - for being friends and partners in research.
- Ms. Ilona Watteler-Spang, Dr. Reinhold Meyer - for the excellent administrative and technical support in our department.
- Everyone who I met during my PhD program.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABBREVIATIONS

BFS	Breadth-First Search
C/S	Client/Server
CSV	Comma-Separated Values
DFS	Depth-First Search
EAI	Enterprise Application Integration
EIP	Enterprise Integration Pattern
ETL	Extract-Transform-Load
FSP	Functional and Structural Plant
groalea	The packages of our interface at client/OpenAlea side
GroIMP	Growth Grammar-related Interactive Modelling Platform
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IMP3D	The 3D graphics library of GroIMP
IOP	Interoperability
JSON	JavaScript Object Notation
K.C. IOP	Four level interoperability of H. Kubicek and R. Cimander
LOD	Level of detail
L-py	Python based L-system
MAppleT	A functional and structural plant model simulating apple tree growth
MTG	Multiscale Tree Graph
ORB	Object Request Broker

PlantGL	The 3D graphics library of OpenAlea
RATP	Radiation Absorption, Transpiration and Photosynthesis
RGG	Relational Growth Grammar
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WSDL	Web Services Description Language
XEG	XML based Exchange Graph
XL	eXtended L-system language
XML	eXtensible Markup Language

Chapter 1

INTRODUCTION

1.1 Motivation

In the early 1990s, process-based crop models emerged as a tool to simulate the development of crops under external (environmental) or internal (biological) conditions. They link the conditions of plants to their overall structural development and provide the possibility to predict the rough growth of the plants. However, the structural or spatial conditions, which play an essential role for plants to maintain their functional conditions, are not considered. Besides, they assume that the conditions applied to every part of the plants are homogeneous and no variation among individual plants and organs is considered. Consequently, they cannot provide accurate growth where plant organs are in the focus and multiple conditions of diversity have to be handled. With the development of computer science, especially the rapid advancement of hardware and software in computer graphics in recent decades, researchers throughout the world have been developing computer models to simulate the complex interactions between three dimensional plant architecture and biological processes that drive plant architecture development in their temporal and spatial contexts [18-21]. These research projects have led to the emergence of functional - structural plant models (FSPM). FSPMs are defined as

models that couple a set of physiological processes that result in an explicit three dimensional plant structure, often supplied with a mutual feedback between physiology and structure [22, 23]. Depending on the application domain, FSPMs have integrated different physical and physiological processes and vary in the level of detail considered for the spatial representation of the plant (considering different hierarchical scales: individual organs, sets of organs or entire plants).

The FSPMs overcome the limitation of process-based crop models by modeling details in function and structure of plants using the increasing computer power, but the cost of complexity caused by these details coming with FSPMs can become prohibitive when they have many processes and depict the plant at relatively fine scale, e.g., organ scale, especially for large plant systems. In addition, such situation also results in a large number of parameters, which makes the indispensable parameter estimation and sensitivity analysis dramatically more difficult as the modeled processes often depend on each other non-linearly.

To solve these problems, four different research groups, including us, have initiated the project "Multi-scale functional and structural plant modeling at the example of apple trees" (i.e. FSPM Apple)[24]. The following introduction is directly adopted from the proposal of the project [25]. Herein, a research agenda with two foci was outlined: F1 – "Model development, calibration, analysis, and corresponding software tools", and F2 – "Case study: Modeling apple tree growth at organ, branch and whole-tree scale".

F1 is about to bridge the gap of complexity between different plant architectures. In F1, algorithms for bridging the gap between spatial and temporal scales (spatial: here organ – branch axis – individual tree – orchard; temporal: hour to year) will be investigated and tools will be established. Methods for upscaling, downscaling and maintenance of multiscale plant representations and processes simultaneously will be developed using open-source modeling platforms.

F2 is a case study to bridge the gap of complexity between two different FSPMs. In F2, an existing empirical, L-system-based model of apple tree growth, MAppleT [26], to which various genotypes and environmental conditions will be applied. Meanwhile, by adopting a prototype for an easy specification and stable solution of differential equations on networks, a xylem and phloem flux model based on biophysics that simulates water and carbon/sugar transport at the branch and organ scale will be established. Such a model will in the future be able to assess the quality of apple fruit under various water conditions. In the end, both lines of work will be combined in an integrated, multiscale model that simulates apple tree growth driven by water and carbon/sugar transport.

My PhD research task is mainly about to enable the case study in F2 and to integrate the MAppleT model and water flux and carbon/sugar transport model. To achieve this, gaps of complexity between different plant architectures inherent in the two models also need to be bridged.

1.2 Research goal and tasks

The research goal is to analyze the requirements and understand the complexity in details, and provide an interface to bridge the gap of complexity caused by the differences between the mechanistic and empirically-based models, and to allow the two FSPMs be merged as one FSPM accordingly.

To reach the research goal, precise tasks have been planned:

- Literature review for FSPM and model integration
- Complexity and requirements analysis of the two different models and their platforms.
- Design of an integrative framework for the interface.

- Implementation of the interface with appropriate technologies according to the designed framework.
- Enhancement of the usability of the interface and the platforms.
- Integration of the FSPMs with the interface and conclusion of the research

1.3 Thesis structure

Chapter 2

In this chapter, theories and technologies in the area of functional and structural plant modeling are introduced. The mainstream FSP modeling approach, i.e. L-systems based modeling systems are discussed in details. This includes the theoretical root of L-systems and the basic technologies required for an effective L-system based FSP modeling system. The various technologies required to enhance the L-systems are also elaborated.

Chapter 3

In this chapter, the theoretical and technical background of the two models and their platforms are introduced. The complexity and requirements of the integration are analyzed in details and preconditions for the model integration are determined. Possible theories and technologies allowing the integration are introduced and discussed, and specific technology candidates are determined according to the preconditions.

Chapter 4

In this chapter, the design of a comprehensive framework for the integration of FSPMs is presented. It includes the design of a middleware technology, a component model, and an architecture that combines a sub architecture adapted

from ETL (Extract, Transform, Load) architecture and a sub architecture adapted from C/S (Client/Server) architecture.

Chapter 5

In this chapter, the implementation of the interface allowing the integration of the two target FSPMs is presented. It includes the design and implementation of *ClientSideInterface* and *ServerSideInterface*, which both consists of a communication group and an ETL group.

Chapter 6

In this chapter, enhancements of the platforms and the interface are presented. For the interface, both functionality and application enhancements are presented. These include the enhancements of applicability, performance, and ease of use of the interface itself, and the enhancements of the application of the interface. For the platform, enhancements for RGG graph usability are presented. Two algorithms for geometrical upscaling are developed and applied to the FSP graphs that are converted from MTGs produced by MAppleT. Several integration applications using the interface are presented and a discussion is presented to conclude the PhD research.

1.4 Functional-structural plant modeling overview

Like the development path of modeling in many other areas, functional-structural plant (FSP) modeling started from a monolithic approach [23]. By this approach, individual FSPMs are built to cover every objective aspects, including design of data models and algorithms, and specific software tools using generic computer technologies. Later, some reusable software components have been developed by plant scientists as common tools to help them to accelerate the modeling speed and reduce the duplicative work. However, these tools normally

come from the practice of solving particular problems in a specific model creation. They cannot provide versatility to suit all kinds of modeling problems. Moreover, these tools are often poorly designed with diverse computer technologies. This makes them hard to be maintained and be used together as a complete tool-set to provide comprehensive support for modeling practice. In the last decade, some teams have started to provide standardized all-in-one platforms providing comprehensive modeling support with well designed tools that suit various modeling cases. After the recent years of development, some platforms have become mature enough. They are widely used in modeling practice and the mainstream approach of FSP modeling is now platform based. These platforms play a role for FSP modeling similar to the role of the development kit for application development, e.g., JDK for Java applications. By providing crucial tools for describing plant systems, the platform is more of a domain-specific infrastructure than just a general development kit. Usually such a platform includes a specific graphics library, a particular modeling formalism built upon a special modeling language with tailored operators and a FSP data model mostly detailed from a general data model (e.g. property graph), some useful components such as 3D viewers and "default" simulators that abstract general functional and structural processes of plants. By this approach, FSPMs are developed and executed on a given platform. As the modeling platform hides all computer-related technical details, plant scientists can thus use the tools provided by the platform transparently to build an FSPM in much shorter time and focus on their own specialty rather than on unfamiliar technologies.

In general, the basic methodology that enables and facilitates the FSP modeling is "encapsulation". The term encapsulation here similar to its meaning in Object Oriented Programming, and refers to the hiding of details of processes. Based on the computer science – biology interdisciplinary nature of FSP modeling [27], the encapsulation can be categorized into two types.

One type is biological encapsulation. This aims at hiding complex biological processes into components so that people without knowing the underlying biological mechanism can directly use them in a way like APIs. At the early stage, the encapsulation was case oriented and happened spontaneously using different computer technologies (e.g. programming languages, FSP data models), abstracted biological knowledge at different levels (e.g. general physiological law VS statistical morphological development patterns based on data measured in a specific region). The outputs were mostly standalone tools with a single or several components incorporating different (composite or primitive) data types of programming languages as FSP data models. These tools were specifically designed for particular modeling cases, can hardly be reused or combined for supporting different modeling cases directly. Later on, some teams carried out systematical encapsulation with a common FSP data model and a set of components abstracting biological knowledge at coherent levels using coherent computer technologies. The outputs became modeling platforms with one data model surrounded by a set of components. The components within a platform thus operate data organized in a platform owned FSP data model in a way similar to transitions operating data in databases. With the systematical and coherent design of the platforms, data operations of the components meet the ACID (Atomicity, Consistency, Isolation, and Durability) properties and the validity of FSP data can thus be guaranteed. Two strategies of biological encapsulation are applied in this kind of platforms design. One is that the components encapsulate biological patterns valid at a specific temporal/spatial/biological range, e.g. only for specific species. Components applying this strategy have high agility and low flexibility in terms of biology. They can be directly used with assignment of parameters, but are only applicable for specific modeling cases. Another is that the components encapsulate general biological laws, e.g., Darcy's law for water transport within plants. Components applying this strategy have high flexibility and low agility in terms of biology. They can be applied for all FSP modeling cases, but only after being extended to suit specific modeling cases.

Another type is computational encapsulation. This aims at hiding complex computational technologies into tools so that people without proficiency in the underlying technologies can directly use them transparently. As a kind of computational model, FSPMs are programs developed in specific programming languages just like all the other computer programs. At the early stage, common computational tools were used directly in FSP modeling practice, e.g., FSPMs were developed directly using common programming languages such as C, C++. This kind of language is not as intuitive as human natural language and needs a relatively long period to master their grammars. More importantly, these languages are not specifically designed for plant modeling, it is difficult to build FSPMs directly by using them, e.g., common programming languages do not provide data models that directly meet the requirement for describing static plant structure and its dynamic growth. To facilitate the FSP modeling, platforms with an adaptive layer on top of the common computational tools to suit FSP modeling have to be established, and theoretically, two types of platforms are possible. One type is that of visual programming platforms (such as early version of OpenAlea [28]), which is about to allow modelers to build FSPMs by manipulating program elements graphically rather than by specifying them textually. With visual expressions, or spatial arrangements of graphic symbols, complex syntax of programming languages becomes transparent, and modelers can concentrate on biological processes/logics design and implementation. Although visual programming brings convenience to modelers, it also has some drawbacks. The logics/processes behind each graphic symbols are predefined with the intention to remain stable, therefore they are with low flexibility whether from an individual or a collective perspective. Compared to traditional textual programming, visual programming FSP modeling platforms provide the possibility to increase the modeling agility but decrease its flexibility in terms of computer science. In this sense, they are similar to component-based platforms applying the first strategy of biological encapsulation. However, unlike the component based platforms, the second strategy is not applicable for the visual programming route because it is about to generalize the logics/processes behind the

graphic symbols and lead into the opposite direction of visual programming. For this reason, this type of FSP modeling platforms does not actually exist. Another type of platforms, i.e., the L-system based platforms, better balance the computational agility and flexibility. Instead of making the grammars of common programming language completely transparent with graphic symbols, L-system based platforms provide formal grammars that are more intuitive than grammars of common programming languages but retain the programming routine, i.e., it is still up to the modeler to develop logics/processes of FSPMs using formal grammars.

Besides the two types, there are also special types of encapsulation, i.e., the hybrid encapsulation. This mainly refers to the component-set based visual programming. This encapsulation increases agility in terms of both biology and computer science, however it still has the problems of low flexibility. The L-systems based FSP modeling approach is what the two target FSPMs adopted, and they are therefore the focus of this thesis.

Chapter 2

FSP MODELING: THEORY AND TECHNOLOGIES

2.1 Functional and structural plant modeling approaches

The name of functional and structural plant (FSP) modeling clearly defines its modeling objective, and makes it easy to be distinguished from other modeling approaches for plants. That is to say, this approach is primarily about to model not only the structure evolving of tangible modules of plants, but also the performance of biological functions of these modules. The main characteristics given by the name are two interactive aspects. One aspect is the interaction between different tangible modules of plants, which describes how one module depends on another and vice versa. Another aspect is the interaction between the structure of a module and its functions, which describes how the structure of a module determines the performance of its biological functions and how the performance of the functions affects or feedbacks to the evolving of the structure. This approach is based on the generally acknowledged truth that the structure of a module is the basis of its

function, and the postulate that a plant is a set of discrete modules (e.g. internodes, blades, fruits) and the set of module types in organisms of one species is finite regardless of the organism size. Thus, the structure evolution of the plant modules is the primary modeling aspect and indispensable part of a FSPM, while functional processes are necessary for being a real FSPM, but they are not an obligation for a plant model, especially when the structure evolution is based on the statistics on growth data of plant structures.

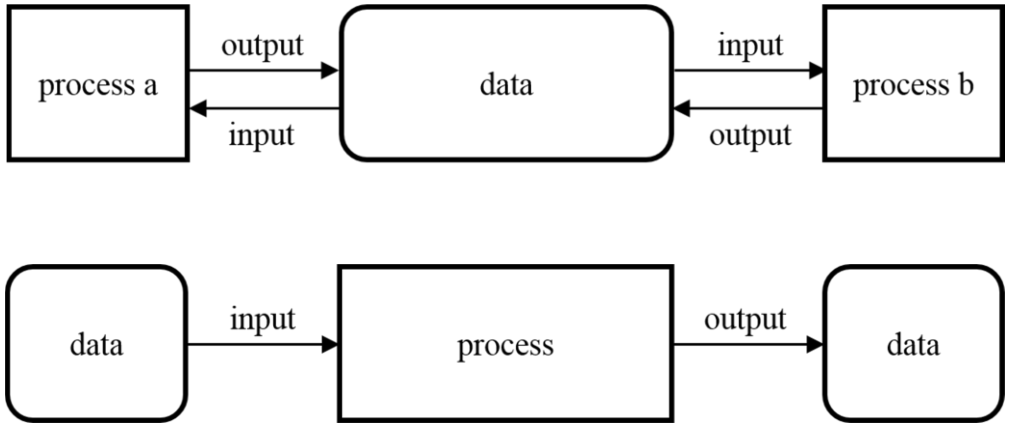


Figure 2.1 Data centric (upper) and process centric (lower) computer programs

A data processing program is a computer program consisting of two basic components: data and processes. As shown in Figure 2.1, these programs can be roughly classified into two types: data centric programs and process centric programs. The data centric program manages data in storage to allow the access and modification by different processes. A process can be launched and transferred to storage when it needs to process data. One example is Apache Subversion. The process centric programs manage processes in storage to allow the data to be input, processed and output. A data package can be input and transferred to storage when it needs to be processed. One example is Microsoft Word.

As a data processing program, a FSPM also consists of data and processes as basic components, and different FSPMs can be divided into two types based on

which aspect it focuses on: the data centric FSPMs and the process centric FSPMs. The FSP modeling methods can be divided into the two corresponding types accordingly.

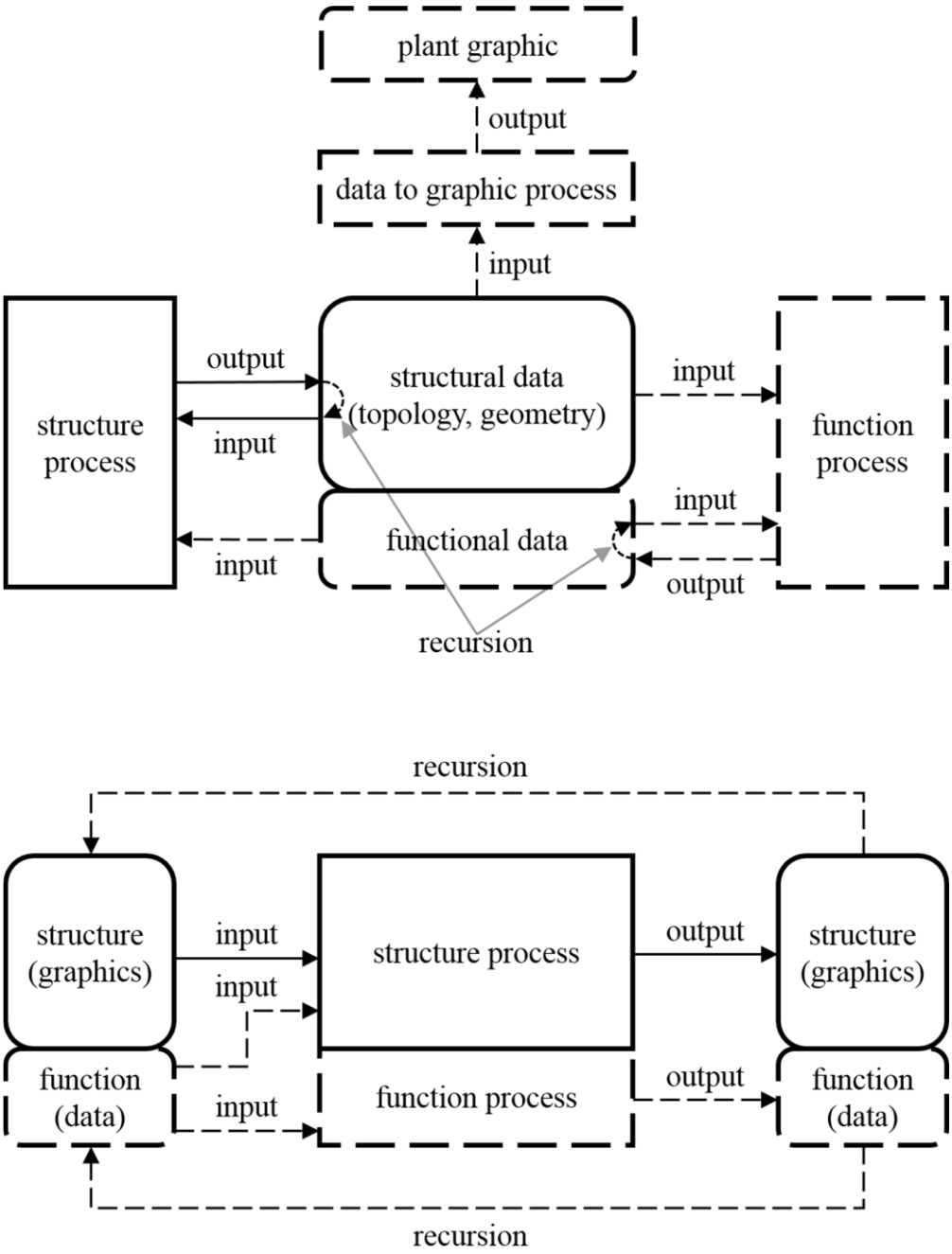


Figure 2.2 Data centric (upper) and process centric (lower) FSPMs.

As Figure 2.2 shows, in data centric FSPMs, the same set of FSP data are accessed and modified by function and structure processes. The data are both inputs and outputs of the FSP processes. An optional graphic drawing (or data to graphic) process can be a part of the FSPM for producing graphic output from the data. The main purpose of the simulation or the execution of data centric FSPM is to compute new FSP data. The production of graphic output is an option. When the processed functional and structural data are accessed and modified by the processes repeatedly, a recursion that represents plant evolving are formed. In process centric FSPMs, structure (graphics) and function (data) are accessed by function and structure processes as inputs and then corresponding results are generated as outputs. The outputs can be again the inputs for further processing, i.e. recursion. The main purpose of the simulation of process centric FSPM is to produce the outputs, mainly the plant graphics. Thus when the graphics are recursively accessed and modified, process centric FSPM can also be regarded as a special case of data centric FSPM with graphics in the center, i.e. graphics centric models.

The next sections of this chapter discuss firstly the basic L-systems for general graphical modeling. Then the two sets of technologies to extend the basic systems to comprehensive systems with two different directions are introduced. One is the L-system extension allowing the construction of processes to produce realistic plant graphics, i.e. extensions for graphic centric plant modeling. Another is the L-system extension allowing the construction of data models to manage FSP data, i.e. extension for data centric plant modeling. Both are introduced and discussed.

2.2 Basic L-systems

2.2.1 Rewriting systems and formal languages

Currently there are several different theoretical frameworks allowing the description of structural evolution of plant modules. The major frameworks are

derived from certain rewriting systems. The rewriting systems, or reduction systems, denote a range of methods to replace sub-terms of a formula with other terms. A typical rewriting system consists of a set of terms/objects and a set of relations to transform the objects. The latter are also called rewriting rules. In general, rewriting systems can be deterministic or non-deterministic. However, the non-deterministic rewriting systems have more than one rule applicable for an object, hence they do not provide a deterministic algorithm for transforming one object to another, but a set of rewriting possibilities.

There are different types of rewriting systems, such as abstract rewriting systems and term rewriting systems. The one where the theoretical frameworks of structural modeling are mostly extended from is rewriting systems operating on character strings, namely string-rewriting systems. Many studies have been carried out on this type of rewriting systems in the middle of the last century. A linguist, Noam Chomsky, made great contributions in this area during his study on formal grammars. He sees languages as formal symbolic systems governed by grammatical rules of combination and defined languages as the construction of words or strings that can be generated using transformational grammars [29, 30]. Basic terms in formal language theory include:

- Alphabet: non-empty finite set of symbols (i.e. letters), denoted by Σ
- Word over an alphabet: finite sequence (i.e. string) of symbols taken from an alphabet.
- Word length $|w|$: number of symbols that compose a word, e.g. $|abcde| = 5$.
- Empty word: the word of length 0, denoted by $\epsilon, \mathrm{e}, \lambda$ or $\wedge$.
- Σ^*: the set of all words over Σ, * is the *Kleene star*, meaning a word has zero or more symbols

- Σ^+: the set of all non-empty words over Σ, $\Sigma^+ = \Sigma^* \backslash \{\epsilon\}$, + means a word has one or more symbols

- The concatenation of two words $v = x_1 x_2 \ldots x_n$ and $w = y_1 y_2 \ldots y_m$ with $n, m \geq 0$ is $v \circ w = x_1 x_2 \ldots x_n y_1 y_2 \ldots y_m$, ($v \circ w$ can be written as vw), the resulting word has length $|v \circ w| = n + m$. If w is the empty word, the resulting word is the original word $v \circ \varepsilon = \varepsilon \circ v = v$.

With the basic terms, the notions of grammar and language are formally defined and we introduce them here by adapting [31]:

A formal language L is a set of words (i.e. strings) over an alphabet Σ, i.e. $L \subseteq \Sigma^*$. The set of all words generated by a formal grammar is a generated (formal) language.

A production rule (or rule, production) is a specification of symbol replacement that can be recursively applied to produce new symbol sequences that conform to the syntax of the language and are composed from the language's alphabet. The grammar of a formal language, i.e. formal grammar, is a finite set of production rules which can generate all symbol sequences (or sentences) of the language.

A formal grammar G is typically defined as a 4-tuple $\mathrm{G} = \langle N, T, P, S \rangle$, with components:

- N: a finite set N of nonterminal symbols

- T: a finite set of terminal symbols, with $N \cap T = \emptyset$

- S: a distinguished symbol $S \in N$ that is the start symbol.

- P: a finite irreflexive set of production rules with the form:

$$P \subseteq \{\langle \alpha, \beta \rangle \mid \alpha, \beta \in (N \cup T)^* \; and \; \alpha \notin T^*\},$$

the production rule $\langle \alpha, \beta \rangle$ is often written as $\alpha \rightarrow \beta$.

Over the alphabet $\Sigma = N \cup T$, for words $v, w \in \Sigma^*$:

- v is directly derived from w (or w directly generates v), i.e. $w \rightarrow v$, if $w = x\alpha y$ and $v = x\beta y$ such that $\langle \alpha, \beta \rangle \in P$.
- v is derived from w (or w generates v), i.e. $w \rightarrow^* v$, if there exist $w_0, w_1, \ldots w_m \in \Sigma^*$ $(m \geq 0)$ such that $w = w_0, w_m = v$ and $w_{i-1} \rightarrow w_i$ for all $m \geq i \geq 1$.
- $\rightarrow^*$ denotes the reflexive transitive closure of $\rightarrow$

Then, $L(G) = \{w \in T^* | S \rightarrow^* w\}$ is the formal language generated by the grammar *G*. The set of all formal languages over an alphabet is uncountably infinite, while the set of grammars generating formal languages over the alphabet with a

Grammar	**Production rules**	**Language**	**Automaton**
Type-0	$\alpha \rightarrow \beta$	Recursively enumerable	Turing machine
Type-1	$\gamma A \delta \rightarrow \gamma \beta \delta$	Context sensitive	Linear bounded
Type-2	$A \rightarrow \beta$	Context free	Non-deterministic pushdown
Type-3	$A \rightarrow \beta B, A \rightarrow \beta$	Regular	Finite state

Table 2.1 The Chomsky hierarchy outlines each of four types of grammars, the form of its production rules, the language it generates, the type of corresponding automaton.

finite sets of production rules is countably infinite. Hence, the set of formal languages generated by a formal grammar is a strict subset of the set of all formal languages.

Chomsky categorized formal grammars and their generated formal languages into a containment hierarchy consisting of four (0 to 3) types of formal grammars over structure conditions on the production rules of the grammars (c.f. Table 2.1) [30]. The hierarchy constrains the structure of the production rules in a restricted set of languages, and the languages types correspond to conditions or the right- and left sides of the production rules [31].

The type-0 grammars are known as phrase-structure grammars or recursively enumerable grammars. They are formal grammars without any restrictions on both sides of the grammar's production rules. Formally, a grammar (N, T, S, P) is a type-0 grammar if and only if all production rules are of the form $\alpha \rightarrow \beta$ with $\alpha \in (N \cup T)^* \backslash T^*$ and $\beta \in (N \cup T)^*$. This type includes all formal grammars and generates recursively enumerable languages that are exactly all recognizable languages by a Turing machine.

The type-1 grammars are known as context sensitive grammars. They are formal grammars with production rules that may be surrounded by symbols (terminal, nonterminal, or empty) as context. Formally, a grammar (N, T, S, P) is a type-1 grammar if and only if all production rules are of the form $\gamma A \delta \rightarrow \gamma \beta \delta$ with $\gamma, \delta, \beta \in (N \cup T)^*$, $A \in N$ and $\beta \neq \epsilon$; or of the form $S \rightarrow \epsilon$, in which case S does not occur on any right hand side of a production rule. Formal grammars of this type generate context sensitive languages that are exactly all recognizable languages by a linear bounded automaton.

The type-2 grammars are known as context free grammars. They are type-1 formal grammars with the left side of production rules restricted to nonterminal symbols and the right side of the production rules restricted to non-empty symbols.

Formally, a grammar (N, T, S, P) is a type-2 grammar if and only if all production rules are of the form $A \rightarrow \beta$ with $A \in N$ and $\beta \in (N \cup T)^*$. Formal grammars of this type generate the context free languages that are exactly all recognizable languages by a non-deterministic pushdown automaton. The context free languages, or more precisely, their subset, the deterministic context-free languages are the theoretical basis of the phrase structure of most programming languages. This type of formal grammars perfectly solves the parsing problem and provides the theoretical basis for the syntax analysis phase of compilation.

The type-3 grammars are known as regular grammars. They are type-2 formal grammars with the left side of production rules restricted to a single nonterminal symbol, and the right side of production rules restricted to a single terminal symbol optionally surrounded by a terminal symbol. Formally, a grammar (N, T, S, P) is a type-3 grammar if and only if all production rules are of the form $A \rightarrow \beta B$ or $A \rightarrow \beta$ with $A, B \in N$ and $\beta \in T^*$, (in this case it is a right linear grammar); or of the form $A \rightarrow B\beta$ or $A \rightarrow \beta$ with $A, B \in N$ and $\beta \in T^*$, (in this case it is a left linear grammar). The type-3 grammars with either right or left regular rules generate regular languages that are exactly all recognizable languages by a finite state automaton. The regular languages can also be generated by regular expressions, which are commonly used for lexical analysis within the scanning phase of compilation.

The incremental constraints from grammars of type 0 to type 4 lead to a directional inclusion relation between the four sets of languages generated by corresponding formal grammars: the set of regular languages $\subseteq$ the set of context free languages $\subseteq$ the set of context sensitive languages $\subseteq$ the set of recursively enumerable languages.

2.2.2 L-systems for graphical modeling

2.2.2.1 DOL-systems

A type of string rewriting systems, the Lindenmayer systems (or L-systems in short) [32-34], was introduced in the late 1960s. Then [2] summaries formal definitions for relevant notations of the L-systems, some of those are introduced here. The simplest class of L-systems is the class of deterministic OL (or DOL in short) systems. A string DOL system G is an ordered triple $G = \langle V,\ \omega,\ P \rangle$, with components:

- V: an alphabet, with V^* denoting the set of all words over V, and V^+ denoting the set of all non-empty words over V, $V^+ = V^* \backslash \{\epsilon\}$
- ω: a distinguished symbol $\omega \in V^+$ that is the start symbol, called *axiom*
- P: a finite set of production rules $P \subset V \times V^*$ with each having the form $\langle \alpha, x \rangle$ or $\alpha \to x$, such that $\forall \alpha \in V : \exists x \in V^* : (\alpha \to x) \in P$. The predecessor and successor denotes the symbol α and word x respectively.

Over the alphabet V, for word $w \in V,\ w = \alpha_1 \dots \alpha_m$ and $v \in V^*, v = x_1 \dots x_m$:

- v is directly derived from w (or w directly generates v), denoted by $w \Rightarrow v$, if and only if $\alpha_i \to x_i$ for all $i = 1, \dots m$.
- v is derived from w (or w generates v), i.e. $w \Rightarrow^* v$, if there exist $w_0, w_1, \dots w_n \in V^*$ $(n \geq 0)$ such that $w = w_0, v = w_n$ and $w_{i-1} \Rightarrow w_i$ for all $n \geq i \geq 1$,
- $\Rightarrow^*$ denotes the reflexive transitive closure of $\Rightarrow$

Then, $L(G) = \{w \in V^* | \omega \Rightarrow^* w\}$ is the formal language generated by the DOL L-system *G*. During each derivation step, all production rules in the set *P* are applied in parallel. Generated by a derivation of *length* n, $w_0, w_1, \ldots w_n$ is called the *developmental sequence* of *w*.

For example, given a string rewriting grammar G with $V = \{a, b\}$, $\omega = a$ and $F = \{a \to ab, b \to a\}$, the grammar (i.e. DOL system) produces strings as shown in figure 2.3.

For a DOL-system, a derivation step produces a string/word representing a plant at a certain growth moment (c.f. Figure 2.3), and a component of the string (i.e. a symbol) represents a plant module. The neighbor relationship between string components represents adjacency between plant modules. The continuous derivation of the system produces a set of strings/words with changed length, which represents the evolution of the plant over time.

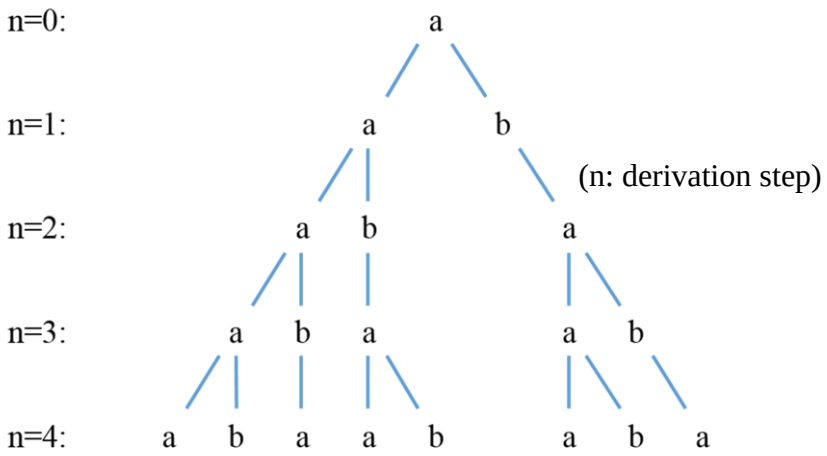


Figure 2.3 Four derivation steps of a DOL-system.

Compared to the rewriting systems, L-systems are outlined by no distinction between terminal and non-terminal symbols and the parallelization of rule application at each derivation step. The former difference reflects the fact that one or more modules or organs of a living body can be dead and lose the ability to

proliferate while the other modules or organs can still support the overall function of the body. Therefore, when using symbols to represent plant modules having changeable states of nonterminal/terminal, a distinction of symbols must not be made. The latter difference reflects a general characteristic of plants that modules of a plant proliferate simultaneously. These differences make the L-systems capable of modeling evolving plant structure, and changes the formal properties of L-systems so that L-systems do not fit into the Chomsky hierarchy.

The studies [35] of L-systems show that L-systems can be is divided into 0L (or OL), 1L and 2L-systems by following the different dependencies in symbol generation of a derivation, i.e. the transition of a symbol depends on zero neighbors, left neighbor and left and right neighbors. An L-system is propagating or a POL-system if there is no production rule producing the empty string ϵ. It is deterministic or a DOL-system if there is at most one production rule applicable for every symbol. As shown in Figure 2.4 [2], the studies [32-37] also reveal the relations between formal languages generated by grammars of the Chomsky hierarchy and those

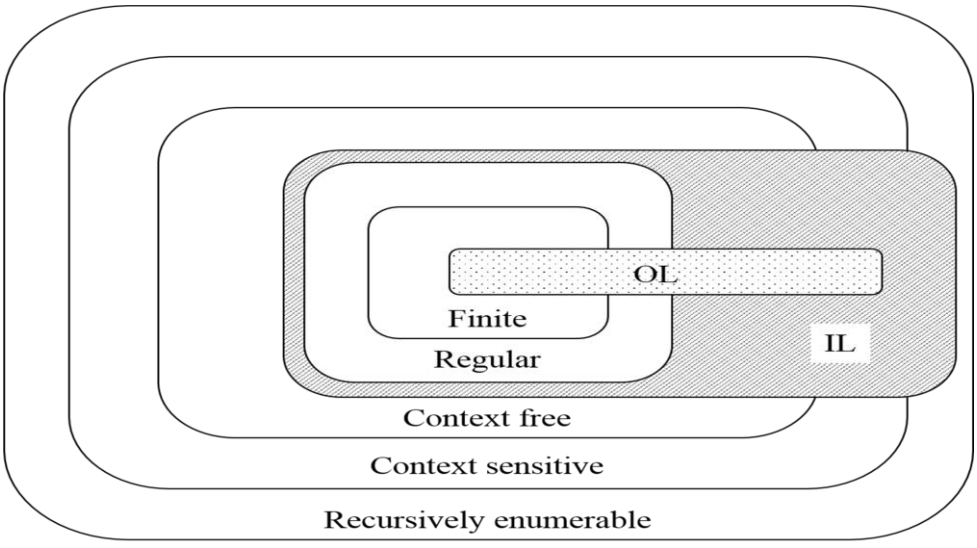


Figure 2.4 Relations between formal languages generated by grammars of the Chomsky hierarchy and the languages generated by L-system grammars [2].

generated by L-system grammars. OL-systems generated languages (OL in short) are not always context free formal grammars, 1L-systems generated languages (1L in short) are not always context sensitive formal languages.

Although L-systems were accepted as a mathematical theory of plant structure evolution, the original version has also the obvious defect that they focus on only on one aspect of the plant structure, the topology (i.e., adjacency between plant modules), without much attention on the geometric side. Therefore, rather than realistic graphics, L-systems can only provide diagrammatic sketches of plants. To make L-systems comprehensive tools for plant structural development modeling, many different geometric interpretations of the systems were studied and proposed. A widely accepted one is the interpretation using turtle geometry.

2.2.2.2 Turtle interpretations of strings

Turtle graphics, or turtle geometry, is a variant of vector graphics using a so-called "turtle" (i.e. a relative cursor) on a *Cartesian plane*. It is a major component of Logo, which is a programming language [38, 39] introduced in the late 1960s. The turtle has three attributes to describe its "current" state including geometrical position (coordinates) and orientation (or rotation) of a virtual pen. The pen also has three attributes to describe its current state including colour and the width the line will be drawn by the pen, and on/off state of the pen. The turtle modifies its geometric state with commands, e.g. "move forward for 9 step length" and "turn right by 30 degrees". Other state variables bound to the pen can also be managed with the turtle, by setting the pen with on/off state, its colour and width. A full turtle graphics system requires procedures that consist of commands, control flow of the commands (e.g. choice or loop) within procedures, and recursion of procedures. From these features, shapes such as triangles, squares, circles and other composite figures can be generated.

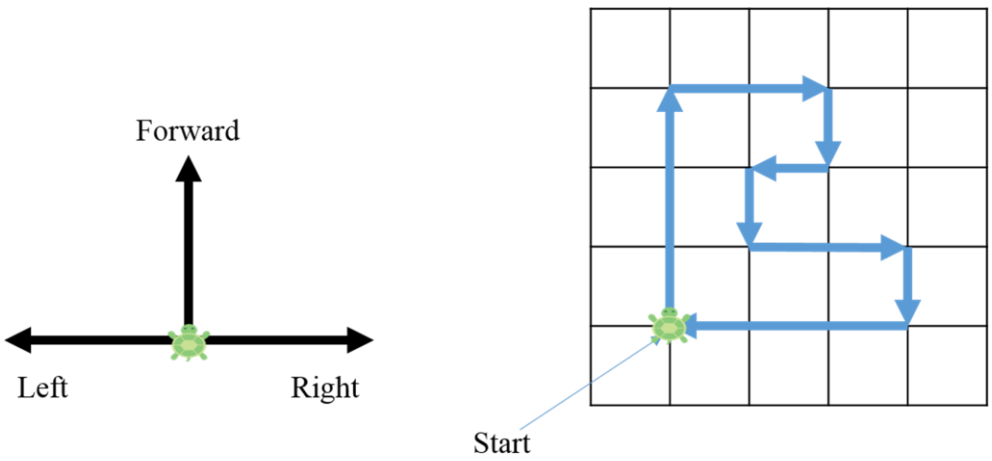


Figure 2.5 (a) Turtle commands F, +, − in two dimensions. (b) Graphical interpretation of a string with fixed rotation angle δ 90 degrees.

From the 1970s on, different graphical interpretation methods of L-systems were introduced (c.f. Figure 2.5). One of them is the turtle interpretation [40, 41], which generates graphics with rigorously defined geometry, such as fractals. Inspired from the turtle graphics of the Logo language, the basic setting of the turtle interpretation includes a *state* variable of the *turtle* which consist of three attributes x, y, and α. The first two attributes represent the "current" position of the turtle in two dimensions, and the last represents its "heading", i.e., angle or current orientation of the turtle forwarding. In addition, a set of commands is also included [2]:

'F': move forward and draw a line from the current position (x, y) to the new position (x', y') by a step of length d. The *state* variable of the *turtle* updates to (x', y', α), with $x' = x + d \cos \alpha$, $y' = y + d \sin \alpha$.

'f': with effects similar to 'F', except no line is drawn.

'+' for turning left by angle δ, the *state* variable of the *turtle* updates to (x, y, α'), with $\alpha' = \alpha' + \delta$.

'−' for turning right by angle δ, the *state* variable of the *turtle* updates to (x, y, α'), with $\alpha' = \alpha' - \delta$.

Based on the basic setting, a string can be interpreted as a graphic drawn by the turtle with given initial turtle state (x_0, y_0, α_0), step length d and angle δ. As a particular kind of strings that are produced by L-systems, this way of interpretation surely works. However, the interpretation in this setting is limited to two dimensions. To allow realistic geometric modeling of plants, further settings to allow three-dimensional geometric interpretation are needed.

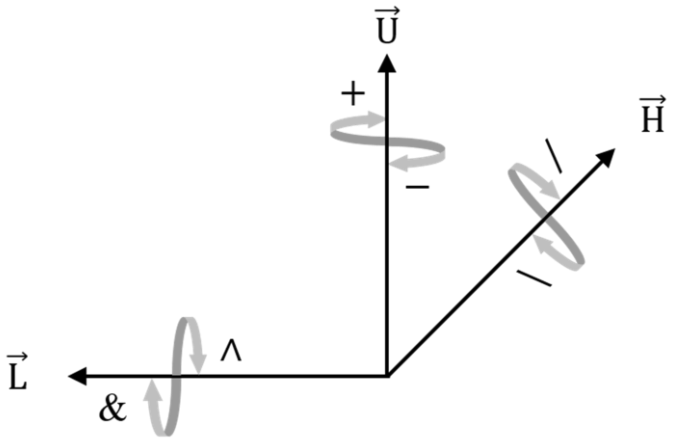


Figure 2.6 Turtle commands in three dimensions

Concepts to allow three-dimensional geometric interpretation [42] have been introduced by Abelson and diSessa (c.f. Figure 2.6). The key is to describe the "current" orientation of the turtle in three dimensions. Three vectors are used to represent components of the orientation on different dimensions: vector H for the heading direction, vector L for the left direction, and vector U for the up direction. It is obvious that the three vectors are mutually perpendicular and normalized, and satisfy the H × L = U equation. With the vectors, a rotation of the turtle updates the "current" orientation of the turtle state from [H L U] to [H L U] R, where R is a 3×3 matrix. Rotations by angle α about H, L and U are represented by matrices:

$$RU(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$RL(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}$$

$$RH(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}$$

The commands +, - defined for two dimensional turtle interpretation keep the same meaning, the matrix RU (α) or RU (- α) is used to compute the new point (x', y', z') of the turtle state from the current point (x, y, z) for a left or right turn by angle α of the turtle. In the case when angle α = 180, RU (180) ≡ RU (-180), therefore the unpaired command | is defined for turning around.

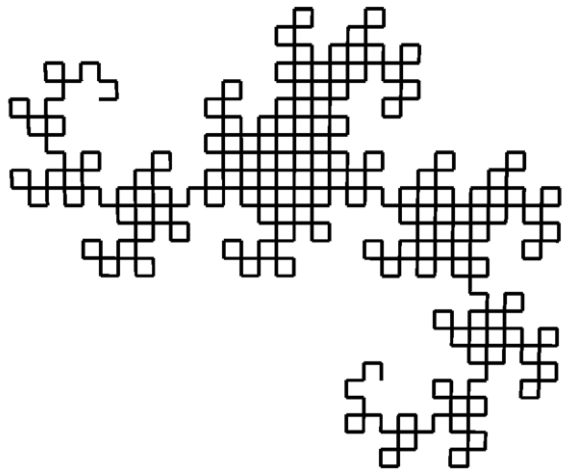
Similarly, the commands & and ∧ are defined for pitch up or down by angle α, the matrix RL (α) or RL (- α) is used to compute the new point (x', y', z') of the turtle state from the current point (x, y, z). \ and / are defined for roll left or right by angle α when the turtle is heading to the H direction, the matrix RH (α) or RH (- α) is used to compute the new point (x', y', z') of the turtle state from the current point (x, y, z).

2.2.2.3 Graphical rewriting

To allow a DOL system to be used for graphically representing the biologically regulated dynamics such as plant structural evolution, figure substitution operations were introduced [2]. Two modes of applying the operations with turtle interpretation were discussed, i.e. edge/node rewriting, with terms originating from graph grammars.

Figure substitution operations capture the recursive structure within figures and link it to a tiling of a plane. The substitution is the combination of strings rewriting

and the turtle interpretation of the strings. Each generated string serves as the input of both rewriting for the next step and the interpretation to generate graphics. For the edge or node rewriting modes, the production rule causes the substitution of figures of new polygons for a polygon edge or polygon node respectively. With the existing setting of DOL-systems, the former mode is possible but the latter is not. To enable node rewriting, symbols representing different subfigures are included into the alphabet of the L-system, and the corresponding subfigures are drawn when such symbols are encountered during turtle interpretation of strings. To ensure the correct position and orientation of the drawn subfigure, a pair of contact points P_x and Q_x are introduced with a pair of direction vectors $\overrightarrow{p_x}$ and $\overrightarrow{q_x}$. (referred as entry/exit points and vectors). With these settings, each subfigure x in a subfigure set X can be correctly appended to the result graphic. During the string interpretation, when the symbol s representing the subfigure is encountered, it will be replaced with the subfigure having its entry point P_x and vector $\overrightarrow{p_x}$ aligned with the current turtle states (i.e. position and rotation). After the placement of the subfigure, the state of the turtle will be updated, i.e. the current position and orientation of the turtle will become Q_x and $\overrightarrow{q_x}$.



Edge rewriting L-system:
alphabet: $\{F_x,\ F_y\}$
ω:F_x, production rules:
$F_x \rightarrow F_x + F_y +$
$F_y \rightarrow -F_x - F_y$

Node rewriting L-system:
alphabet: $\{x, y, F\}$
ω:Fx, production rules:
$x \rightarrow x + yF +$
$y \rightarrow -Fx - y$

Figure 2.7 The same dragon curve generated by edge and node rewriting L-systems with n=9, δ=90

The graphics generated by the edge and node rewriting systems are not disjoint, and sometimes an edge rewriting system can be transformed into a node rewriting system using a pseudo L-system as bridge, namely by introducing a predecessor containing more than one symbol, so that a substring may be substituted by the successor of a rule. Figure 2.7 shows the same dragon curve [43] for generated by edge and node rewriting L-systems with nine derivation steps. The alphabet of the edge rewriting L-system includes two different symbols representing the same turtle command "move forward a step by a specific length", while the alphabet of the node rewriting L-system uses two symbols representing the "subfigures that are reduced to single points". The edge rewriting system shown in the figure can be rewritten as a pseudo L-system with the non-turtle interpretable symbols *x* and *y*: $\omega: Fx, P: \{Fx \rightarrow Fx + yF+,\ yF \rightarrow -Fx - yF\ \}$. From the pseudo L-system [40], string rewriting rules from *x* to *x*+*yF*+, and from *y* to *–Fx*-*y* can be found, thus it can be transformed to the node rewriting system shown in the figure.

2.3 L-system extensions for graphic-centric plant modeling

Graphic centric plant modeling is about using L-system strings to represent plant structure. Combining the turtle interpretation of strings, the technical basis of this modeling method is the graphic rewriting. The plant graphic drawing is based on the turtle interpretation of the strings.

2.3.1 Plant topology modeling

Compared to the string rewriting L-systems, the graphical rewriting L-systems have already a certain strength for modeling plant structure graphically. However, the modeling is still quite limited. On one hand, the generated graphics can only be linear with all modules in a sequence. On the other hand, the generated graphics is

considered as a single structural unit without capability to express different types of adjacency between graphic components representing plant modules.

In real the world, the structure of plants normally consists of branching structures with plant modules connected in different topological types, such as trunk or branch modules. Moreover, each plant module (organ, tissue…) has certain functional roles, e.g. a blade plays a role in photosynthesis, and an internode plays a role in water transport. These modules directly or indirectly depend on each other, i.e., it is only possible to maintain the normal function and structure of a module if the function and structure of other related modules are normal. Consequently, when different plant functions are involved, the research is on module (organ, tissue…) scale and it is more appropriate to graphically represent a complete plant with a combination of multiple structural units representing plant modules rather than a single unit.

To express the multi-module composite branching structures, another extension of L-systems including rooted tree based axial trees [44, 45] and tree OL-systems was introduced. The **rooted tree** [46] comes from a mathematical notion from Graph Theory, where mathematical structures are established to model pairwise relations between objects. A rooted tree consists of a set of edges and a set of nodes just like other types of graphs. What makes it special is that it has a tree-like structure without cycles. In the structure, edges are labeled and directed. A special node “root” is distinguished and all the other nodes are connected from it by edges directly or indirectly.

By adding a collection of additional topologic specifications to a rooted tree, a special type of rooted tree, the axial tree, was introduced to allow the expressiveness of branching structures in L-systems. As Figure 2.8 [2] shows, in an axial tree, each node has at most one outgoing *straight* edge, and all the other non-straight edges are referred as *lateral* or *side* edges. A totally ordered set of *straight* edges forms an *axis* with the condition that the first edge is lateral or originates from the root

and no straight edges follow the last edge. An axes and all its sub axis form a branch or a sub tree. Depending on the level of nesting, an order number is given to the axis and branch. The axis with first edge originating from the root has order number 0. An axis with first edge being lateral and the source node of this edge belonging to an axis with order number n has order number n+1. Beside the topological expressiveness at organ level, the edges in the rooted tree (or different branches in the axial tree) represent real plant modules in exactly the same way as in L-systems with simple turtle interpretation. Hence, the axial tree gives L-systems expressiveness of both topology and geometry at organ level. It is worth noting that the application of topology or the rooted tree is not the same as it is in Graph theory or Data models, for example, here they are borrowed for explanation of the arrangement of graphics, while in Data models they are used for describing the arrangement of data elements.

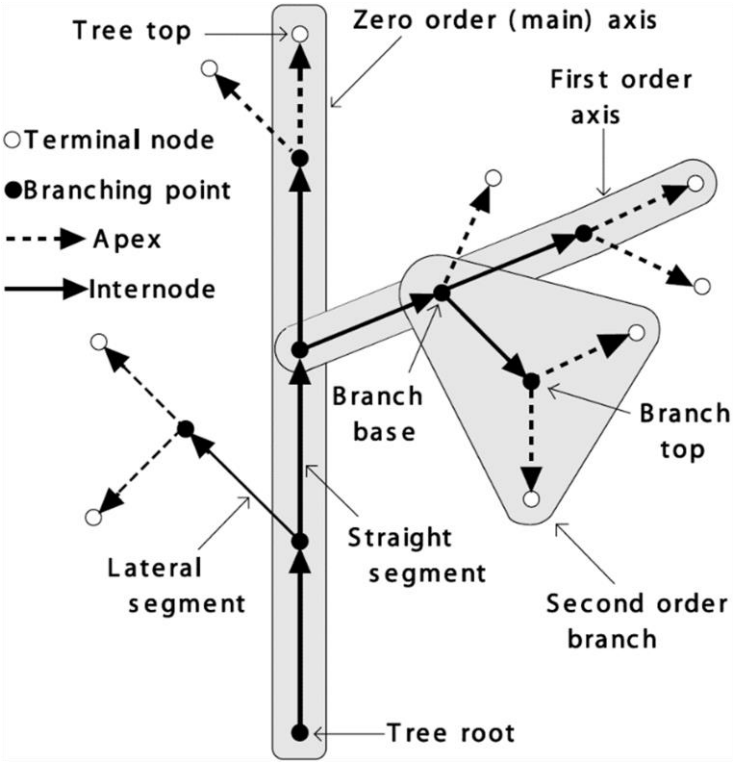


Figure 2.8 An axial tree [2]

Meanwhile, OL-systems evolved to **Tree OL-systems** to allow the modeling of the development of axial trees with branching systems, i.e. rewriting of axial trees. Similar to the DOL system, a tree OL system *G* is defined by three components:

$$G = \langle V,\ \omega,\ P \rangle$$

V is a set of edge labels and ω is an initial tree with labels from the set. *P* denotes a finite set of tree production rules. Figure 2.9 [2] shows a tree production rule and its application. With an initial tree T_1, a given tree OL-system generates a new tree T_2 after applying the production rules once.

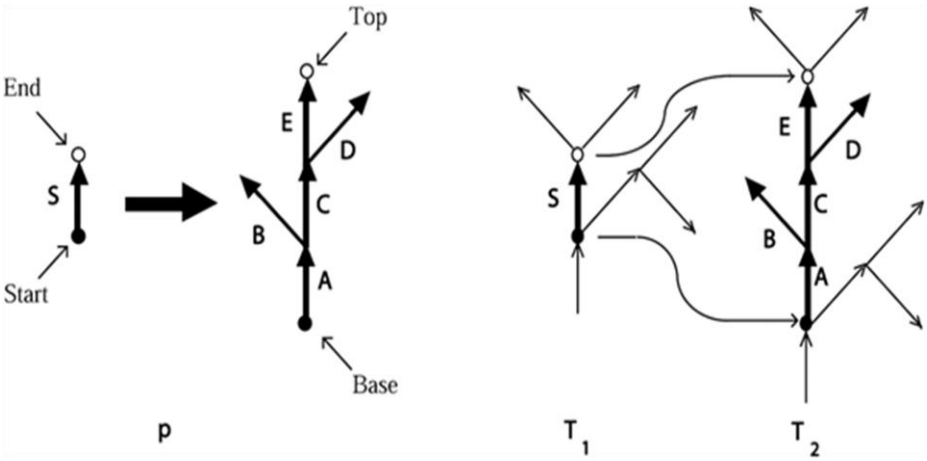
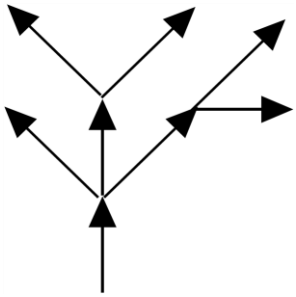


Figure 2.9 An example of applying a rule P to the edge S of an initial tree T1. [2]

The evolution to tree OL-system alone is not enough, and relevant supporting measures are also needed, namely specific grammatical settings denote different topological types. Otherwise, production rules P will only be expressed in a form similar to the one in Figure 2.5, and the turtle interpretation of a branch will always need to start from the root. To make up for this deficiency, tree OL-systems were again extended to bracketed tree OL-systems. New grammatical components, brackets "[" and "]" were introduced, so that the production rules can distinguish the edges of a branch. Moreover, this allows the *state* of turtle at the starting point

of a branch to be stored in a stack, so when the interpretation of one branch with order n+1 is finished, the “turtle state” can be used for continuous interpretation of the axis with order n (where the branch is originating). Figure 2.10 [2] shows the string representation of an axial tree using the concept of bracket.



F[+F][-F[-F]F]F[+F][-F]

Figure 2.10 An example of representing a tree by a bracketed string [2]

2.3.2 Plant geometry modeling

The enrichment by plant topology enables the topological expressiveness of graphical rewriting L-systems, while the problem of lack of geometry expressiveness remains, i.e. the graphics generated by production rules are still nonrealistic compared to plants in nature. This problem is solved by providing continuous and composite geometry within the graphics.

2.3.2.1 Continuous geometry

One reason for lacking realism is the turtle command *F* drawing a line segment with fixed length. The result of this setting is that the length of every plant module represented by a line segment is the same, even if a plant module is represented by a group of line segments, the length is still limited to an integer multiple of a fixed length. Hence, it is impossible to have a plant module with continuous length. Not to mention the size of the corresponding string will be rather long when a large plant is addressed. On the other hand, the length should not be fixed when functional effects are taken into account. That means the length might be computed with a

coefficient that represents the effect of certain functions. To allow an adjustable geometry (e.g. step length, rotation angle), the L-system symbols (e.g. turtle commands) associated with parameters, i.e. the paramedic L-systems [2, 47, 48] are introduced . As the parameters are not fixed values, the computed result, i.e. adjusted geometry, is continuous.

The turtle interpretation of parametric words is included into the L-systems accordingly. The principle is to take the first parameter to control the corresponding turtle state, default values are applied if there is no parameter. The main parametric symbols include [2]:

'F(a)': move forward and draw a line from the current point (x, y, z) to the new point (x', y', z') by a step of length a. The *state* variable of the *turtle* changes to (x', y', z'), where $x' = x + a\,\overrightarrow{H_x}$, $y' = y + a\,\overrightarrow{H_y}$, $z' = z + a\,\overrightarrow{H_z}$

'f(a)': with effects similar to 'F(a)', except no line is drawn.

'+(a)' rotating around $\vec{U}$ by an angle of a degrees. Depending on if a is positive or negative, the turtle is rotated to the left or the right.

'&(a)' rotating around $\vec{L}$ by an angle of a degrees. Depending on if a is positive or negative, the turtle is pitched down or up.

'/(a)' rotating around $\vec{H}$ by an angle of a degrees. Depending on if a is positive or negative, the turtle is rolled to the right or left.

For example a production rule that multiplies the length x of an internode I in every derivation step by 1.2 can be written as $I(x) \rightarrow I(1.2 \times x)$.

Besides fixed length, all line segments have the same width as well. This problem can be solved by simply providing different types of line segments with

different widths or by using another parameter representing the width of line segments.

2.3.2.2 Composite geometry

Another reason for lack of realism is that the production rules can only generate graphics consisting of line segments and no shapes are generated to realistically represent the real plant modules. The problem is solved by providing a mechanism similar to the “subfigure” introduced for node rewriting with predefined figures of polygon shapes [2]. In detail, surfaces are composed bicubic patches defined by polynomials. Symbols representing different surfaces are included into the L-system alphabet, and during the interpretation of the strings produced by production rules, the turtle draws the surface when its symbol preceded by a tilde was detected. Beside the shape itself, the position and orientation of the surface is determined similar to that of a subfigure (using contact points and vectors).

2.4 L-system extensions for data-centric plant modeling

Data centric plant modeling is about using data models (i.e. graphs in the context of FSPMs) to represent plant structures. Generalizing the L-system production rules, the technical basis of this modeling method is the graph rewriting. Instead of turtle interpretation, the plant graphics generation is based on the combine of graph traversal and turtle interpretation. Compared to graphical centric plant modeling, the essence of the data centric modeling is organizing FSP data in data model. There are two main advantages to do so. The primary one is automatic management of the dependencies between different kinds of data and data of different plant modules. The secondary one is plants can be modeled by a structure more general than tree, e.g. a multiscale structure, where a coarse scale node decompose to two

successively connected fine scale nodes, is a circle, which cannot be modeled by a string but a graph data model.

2.4.1 Graphics library

The plant graphical rewriting L-systems with graphic centric extensions enable realistic graphical modeling of plants, but it has the limitation that the strings are directly interpreted as graphics, and geometry data are not stored for possible further usage such as functional processes by considering geometry. To make geometry data of a plant module reusable, the concept of data structure must be applied. That means each plant module should be associated to a variable of a specific graphic type. When relevant data processes are required, access and modification are all through the variable, and so data are kept available for further processes after finishing the current process. The key here is to have a suitable graphic technology to provide data structures for both geometry data management and precise graphical representation of the geometry.

To allow a graphic data structure, there are actually two different two technical paths: vector graphics and raster graphics (c.f. Figure 2.11 [6]). Vector graphics

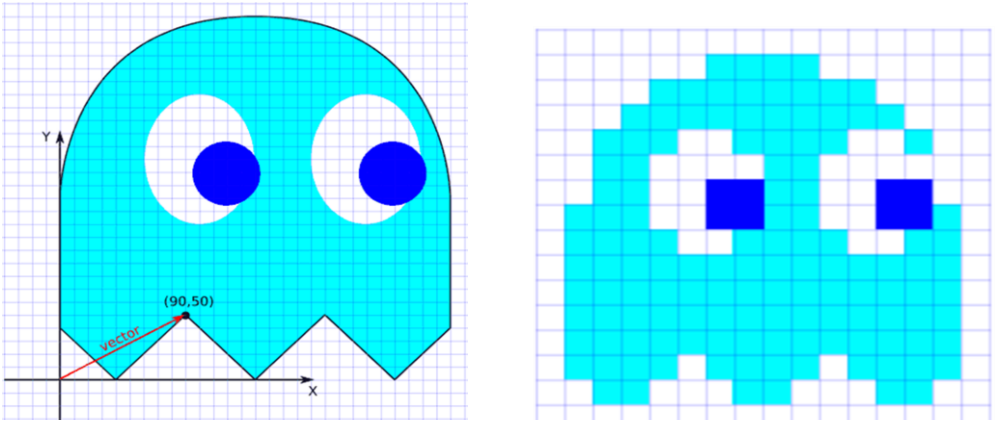


Figure 2.11 Examples of vector graphics (left) and raster graphics (right) [6]

[49-54] use mathematical formulas to describe graphics, usually showing things such as lines, curves, and shapes, which can be defined by vector graphic types. Raster graphics [52-54] describe graphics as a series of color values, which are then placed in grid mode as basic unit of a raster graphic, i.e., pixels in 2D and voxels in 3D graphics. Both technologies have capability of managing data and presenting precise graphics, but the mainstream technical path in the area of FSPM is vector graphics [2, 12, 40, 44, 55-58]. In fact, compared to raster graphics vector graphics has some advantages of higher weight.

One advantage is that the controlling and computing of geometry for vector graphics (through parameters) are more straightforward and accurate. Another advantage is, regarding storage requirement, that raster graphics has a high demand as it stores information of the grid for not only target objects that are normally tangible but also the empty space around them, while vector graphics has a low demand as it stores only the mathematical formulas and the parameters for target objects. Additionally, vector graphics has the capability of undistorted scaling while raster graphics has not.

Above all, the fundamental reason for L-systems to take the vector graphics over raster graphics lies in the natures of the two technologies. In fact, in the bracketed tree OL-systems, the turtle command F represents both graphical transformations and drawing of line segments and other turtle commands represent only graphical transformations. In the context of including graphical representation of organs, the line segments are expected to be replaced with graphical shapes, thus the turtle command F needs to be replaced by different graphical objects. Therefore, it is logical that each of these graphical objects represents both transformation and graphical shape. For example, a Cylinder object not only needs to have the capability to draw a cylinder shape, but also needs to have the capability to change the turtle state from “start position” to “end position” of the object. Hence, both capabilities need to be enabled as a part of new L-systems and this can be done for

both raster and vector graphics theoretically: Raster graphics drawing can be enabled by setting up a set of specific raster primitives using certain encoding technologies, e.g. bitmap, jpg, or png, for raster graphics. While vector graphics drawing can be enabled by setting up a set of vector primitives using different generic vector graphic libraries[53], e.g. OpenGL. However, the raster graphics is discretized and is normally used in visual input (e.g. optical/digital photography or laser scanning) or visual output (e.g. inkjet/laser printing or CRT/LCD/LED displaying), while the vector graphics is continuous and is normally used in graphic computing [52, 54]. The main reason of the division of work comes from the natures of the two different types of graphics: technologies for input and output are all discretized sampling based, while technologies for graphic computing are all continuous geometry based.

In addition to being of vector graphic type, the function as a turtle of an L-system needs to be guaranteed as well. This includes the definition of the start position and end position of the turtle for the specific graphic type, the local transformation for the change of the turtle state, and the method to execute the transformation. The last two are based on the first definition, and they are not always easy, especially for non-convex vector shapes (e.g. NURBSSurface). Fortunately, non-convex shaped organ barely locate at non-terminal position in nature, thus it is not necessary to define them as turtle actions. In many cases, it is more appropriate to define them as an independent library of “non-turtled” vector primitives that complements the library of “turtled” vector primitives rather than having a library that mix them together.

Besides the turtle command F, other turtle commands, such as “RU”, “RL”, and “RH” are also needed to be a part of the library as well. It is clear that these types are just pure turtle commands, they will not be used for representing organs.

In the last decade, some 3D graphics libraries were introduced to make up for this lack for L-system specific constructions. Due to previously described reasons,

these 3D graphics libraries are mostly of vector graphics and consist of a rich set of well-defined vector primitives that are tailored for FSPM, e.g. IMP3D of the GroIMP platform. As a part of the L-system, these primitives are not only vector shapes or transformations but also turtle commands.

The usage of a vector graphics library fulfills the needs of having geometric data structures and reflects the evolution from graphics to data structure. On one hand, graphical representation of the geometry is ensured by the inclusion of both shape and transformation types. To make the usage of the library flexible, the library includes not only basic transformation types, e.g., Translation, Rotation and Scaling, but also different transformation matrix types, e.g., Matrix4d, Matrix34d. On the other hand, the types are essentially data structures for organization of geometry data. In detail, they are mostly implemented using the class concept of object orient programming. Different classes represent different graphics types, which include data fields coded in typed variables and their applicable operations coded in methods. After including the 3D graphics library, the L-system alphabet needs to include symbols representing geometric instances while the symbols representing subfigures can be retained.

2.4.2 FSP data model

2.4.2.1 Notions of data structure & data model

The concepts and notions of data structure [59, 60] and data model [61] are basis of the data centric plant modeling, they contain the fundamental reason why the advancement from graphic centric plant modeling to data centric modeling is necessary for FSP modeling, and why a FSP data structure & data model is in the center of the modeling. These notions [59-62] are interrelated but different, thus are much confusing and should be compared and understand first.

A **data** is a symbolic representation of an objective thing. In computer science, it refers to a sequence of one or more symbols given meaning by specific act(s) of interpretation.

A **data type** is a classification of data. It includes a class of data with certain similarity (e.g. precision), namely a collection of values, and the operations that can be done on the data. The data type defines the meaning of the data, and the way values of that type can be stored. Data types of a high level programming language can be divided into atomic (or primitive) types with indecomposable values, e.g. integer or boolean, and aggregate (or composite) types with values aggregated (or compound) in a certain way (i.e. decomposable values), e.g. list or array. An **abstract data type** (ADT) defines the blue print of a data type by a mathematical model with applicable operations.

A **value** is a data with a given type. The members of a type are the values of that type. When a data is classified into a type, it becomes a value and can be manipulated by a program. Data and value can also be distinguished using the data structure concept (c.f. next page): a data is unstructured and a value is structured, when a data is structured, it becomes a value. When certain meaning or interpretation is given to a value, it becomes an **information**. The meaning or interpretation is understood as **semantics** of the information, while the data structure is understood as **syntax** of the information.

A **literal** is the representation of a fixed value in source code. Most programming languages allows literals of both primitive and composite data types, such as integers and arrays.

A **variable** is a symbolic name (an identifier) bound to a storage location holding a changeable value. It allows the name to be used independent of the value it represents. The variable can be bound to a value during the compiling or run time. The symbolic names of variables are a usual way to reference the stored values, and

are replaced with the actual storage location of values by compilers or interpreters. Values in locations change during program execution while locations and names are fixed. A **constant** is a special variable with value does not change during program execution.

A data structure is a specific way of data management, i.e. organization and storage of data to allow its efficient access and modification. A data structure is the implementation of one or more abstract data types (i.e. an actual data type), and consists of a collection of typed data (i.e. values) referred as data elements, the relations among them, and the functions or operations that can be applied to the data. The data element is the basic unit of data and is usually considered and processed in the computer as a whole. Sometimes a data element can consist of several typed data referred to as data items. Data items are the indivisible minimum units of a data structure.

According to the abstract description method and the internal storage form, the data structure can be divided into logical structure and physical structure. The **logical structure** describes the logical relations between data elements in a data structure using an abstract mathematical model. The **physical structure**, also known as storage structure or storage image, is a storage representation of a data structure in primary storage (or main memory) or secondary storage (or external memory) of a computer.

A data element stored in memory is also called a node, and each data item in a data element is called a data field. Nodes can be seen as storage structures of data elements, which are represented by bits in certain memory units. The logical relations between the data elements are represented in the computer by a sequential, linked, indexed, or hashed image, in *sequential*, *linked*, *indexed*, or *hashed* storage structure.

Data elements can be organized in different structures, which can be roughly divided into four basic types. (1) *Set*: There is no other relations between data elements in the structure other than the "belong to the same set" relation. Structures of this type are usually represented in the computer as hashed images. (2) *Linear structure*: There are one-to-one relations between data elements in a structure. Structures of this type are usually represented in the computer as sequential or linked images. (3) *Tree structure*: There are one-to-many relations between data elements in a structure. Structures of this type are usually represented in the computer as linked images. (4) *Graph or Network structure*: There are many-to-many relations between data elements in a structure. Structures of this type are usually represented in the computer as linked images.

The data management of a data structure is based on the ability of a computer to fetch and store data elements in its memory by address. Specifically, data management for the sequential data structures is based on computing of the address of data elements with arithmetic operations, while data management for the linked, indexed, or hashed data structures is based on storing address (i.e. pointer), index, or hash of the data elements within the structure itself.

The most widely used data structures [60] include:

- Linear data structures, including linear list, linked list, stack, queue and array, in which one data element has at most one direct successor or predecessor, i.e., the one-to-one relations are directed.

- Tree data structures, including tree, binary tree, in which one data element has one direct predecessor but more than one direct successor, i.e., the tree data structure is hierarchically directed and without cycle.

- Graph data structures, including directed graph and undirected graph, in which data elements have many-to-many oriented or non-oriented relations.

A data model defines the schema how the elements of data are organized and interrelated, and how they are related to properties of entities in the real world. It is a concept from the perspective of application and is occasionally referred to as data structure from the perspective of technology, especially in the context of computer languages. A data model can be conceptual, logical or physical [61, 62]. A *conceptual data model* gives only general high-level data constructs with no technical terms. It allows business information to be captured in non-technical way so that the technical designer can take over the conceptual data model from the business designer to do further logical design. A *logical data model* gives detailed data structure using technical terms, such as tables of the relational data model, classes of object-oriented data models, or tags of XML based data models. A *physical data model* gives the implementation of a logical data model using specific technical tools such as specific database management systems for relational tables, specific object-oriented programming languages for classes. Besides the definition of "*How* (data elements are organized)", the term data model sometimes refers to a set of concepts (e.g. entity, attribute, relation in the ER model) used in constructing such schema, i.e., "*What* (is used to do so)".

Data models precisely describe the objective natural system's statics, dynamics and integrity constraints. Thus, a data model normally includes a data structure[1] for static descriptions, data operation for dynamic descriptions and integrity constraints.

There are many types of data models, including:

- Database model, describing how data is managed in database.
- Data structure diagram, describing conceptual data models through graphical notations that describe entities with their relationships and relevant constraints.
- Entity-relationship model [63], describing interrelated things of interest in a specific domain of knowledge. A basic ER model is composed of

entity types (which classify the things of interest) and specifies relationships that can exist between instances of those entity types.

- Generic data model, defining standardized general relation types, together with the kinds of things that may be related by such a relation type to facilitate data exchange and integration.

With its logical structure or schema, a data structure is capable of physical storage image manipulation, while a data model is capable of the real world expressiveness. As the focus of this thesis is not the computer technology itself but the application of the technology, the introduction and discussion does not focus on data structures but data models, and in this section, we discus only the data model at conceptual level. Different concrete plant data models at logical and physical level will be introduced and discussed as a part of the design of our architecture for the integration of different functional and structural plant models. Moreover, this thesis mainly discusses the "How" aspect of the data model, therefore the first type (i.e. database model) is in our focus.

The most widely used data models (or, more specifically, database models) [64-68] include:

- Hierarchical model, in which data is managed in a tree structure. The data elements are stored as records which are related to one another by links. It allows one-to-many relationships. A record is a group of correlated fields, where each retains a single value. The entity type of a record specifies which fields the record holds. The record and entity type of a hierarchical model respectively correspond to the row (or tuple) and table (or relation) in the relational model.

- Network model, which expands upon the hierarchical structure, allowing many-to-many relationships in a graph structure with the possibility of

multiple parents. It operates at a low level of abstraction and lacks easy traversal over a chain of edges.

- Relational model, in which data is managed into a structure conforming to first-order predicate logic and set theory, with all data being held in tuples, which are then grouped and stored into relations, namely two-dimensional tables.

- Graph model, in which data is managed into a graph structure where one node may be connected to any other node. Although the structure is the same as that of the Network model, the Graph model has a clear separation between the model and the actual implementation and it is easy to traverse over a chain of edges, which makes semantic queries with nodes, edges and properties possible. Within the graph structure, data elements are directly related through relationships (i.e. edges) in the data store so in many cases they can be retrieved with one operation. The graph data can be stored differently, for example, "into" relational tables with an additional level of abstraction, i.e., regarding the tables as nodes and edges of the graph; or into key-value based structures such as *dictionary* or *hash*; or document based structures such as *XML* or *RDF*.

Depending on the degree that the schema constrains data, data models can be divided into *structured* and *semi-structured data models*. The structured data models have a clear separation between schema information and data. The schemas fully constrain data and are normally predefined with the intension to keep them stable. The semi-structured data model is a data model having the schema information mostly contained within the data, i.e., without predefined schema that is separate from data. In some cases, schema information is contained within a predefined "weak" schema (with only a few restrictions) that is separate from data. Often, the semi-structured data model is referred to as self-describing data model. In general, the use of structured or semi-structured data models depends on whether

the data-modeling object and objective is generic or specific. For example, if the data modeling object is a student, and the objective is the student registration management, then the schema can be predefined as precisely as possible. In case of FSPM, a structured data model is good enough for a specific plant species, while a semi structured data model as a basic feature of a FSP modeling platform provides better flexibility for various modeling use cases of different plant species.

2.4.2.2 Modeling of FSP data

2.4.2.2.1 The necessity and requirements

The addition of a graphics library to graphical rewriting L-system solves the problem of unmanaged geometry data. However, it is still difficult to model the function of each plant module and the structure – function interaction within and between organs. The essence of interaction between structure and function of a plant module is that the computation of new structure takes not only existing structural data but also existing functional data into account, and vice versa. The essence of interaction between different plant modules is that the computation of new FSP data of one plant module takes not only the existing FSP data of itself but also the existing FSP data of other related plant modules into account.

In graphical rewriting L-systems, literals and variables with primitive data structures are used as parameters to compute continuous geometry values of a plant module. Using variables of primitive data structures, functional data of a plant module can be taken into account for the computation of geometry values. However, modelers have to manually ensure that the functional data for computation of geometry values is of the right plant module. Moreover, the resulting geometry values of the computation of plant modules are literals without means to reference the stored values to allow further computation of functional data, i.e. structural feedback to function. Besides that, the two different topology relations between plant modules are distinguished by the appearance of bracket symbols, these symbols are interpretable by the turtle for graphic drawing but are not able to

take into account (as arguments) the interaction of geometry values and functional data between two plant modules for computation. The addition of a graphics library already allows the resulting geometry values to be used for further computation of functional data. But the other problems still remain unsolved.

The main reason for the defects is the lack of a uniform data model in the system that manages functional and structural data of a plant module together and the data of different modules together. This lack causes a separation of functional and structural data for each plant module, and a separation of the data of one module from the others. In other words, the problems can be solved only when the L-systems advance from graphical rewriting to graph rewriting.

On one hand, as the structural data mainly refer to the shape and location of a plant module, the related data are managed as fields of an instance of a graphic primitive, which are predefined as a part of the library/L-system and do not depend on specific modeling cases. In contrast, functional data refer to plant function, which depends on specific modeling cases, thus it is impossible and not logical to predefine them as fields of a graphic type. Without a uniform data model, FSP modelers have to manage the link between functional data and corresponding structural data of a specific plant module. This makes the modeling of the function and the interaction between function and structure of a plant module very difficult.

On the other hand, because of the same reason, the structural data include only the quantitative aspect (i.e. the shape and location, or the turtle state), the qualitative aspect (i.e. different adjacency between plant modules) is not managed at all. Actually, the neighbor relationships between modules has been considered in to axial tree, but that improvement was made only to produce branching graphics. Although the neighbor relationship in axial tree seems acquirable by string scanning in theory, it is difficult for modeler in practice. The separation of the model and the modeling platform is to free the modelers from the complex technical work, so that they can focus on their areas of expertise. However, such acquisition requires the

modeler to be familiar with platform-level technologies and know how to manipulate L-system string. In fact, the key defect of axial tree is that the distinguished two different neighbor relationships (lateral or axial) does not managed by a FSP data model and thus cannot be used for finding out the related modules by applying the platform level technologies. Theoretically, it is not possible and also not logical for adjacency to be managed as the fields of an instance of graphic primitive because they depends on specific modeling cases. Therefore, FSP modeler have to manage the link between specific adjacency and corresponding structures representing source and target plant modules. This makes the modeling of the interaction between function and structure of different organs very difficult.

Essentially, the FSP data of parametric L-systems with library are literals, or individually managed constants/variables. This might be suitable for simple abstract and conceptual FSPM development, but not for high-precision modeling tasks that include complex details. In order to allow FSPM with accurate details and to free the modeler from heavy technical work, FSP data models have been designed and introduced as a part of rewriting systems.

2.4.2.2.2 Conceptions of FSP data model

From the perspective of FSP modeling application, or the point of view of FSP modeler, a logical data model is needed to bridge the gap between real world plants and digital plants expressed by a physical data model, while from the technical perspective of an FSP modeling system, a logical data structure is needed to allow the data management at storage level by operations at logical level.

The intersection part between the needed logical data model and the logical data structure is the logical structure of data elements. Through it, real world plants are physically represented and the relevant storage image is physically managed. Thus, it is both a data model and a data structure. Nevertheless, we refer to it as FSP data model because the expected logical structure cares more the expressiveness in the

specific domain of FSPM than for the effectiveness of data management, and the description of the logical structure essentially reflects the underlying structure of the domain itself, thus it is overall not technology oriented but application oriented.

The FSP data model refers to two levels of data management. One is the data management at the plant module level. At this level, the same data model should manage all kinds of data relevant to a plant module. Which include geometry data and functional data, such as length/width, water pressure or sugar content. Simple usage of a graphic data structure does not meet the demand. The relation between the graphic data and functional data is obviously that they belong to the same set, thus a data structure that simply composes the graphic data structure and functional data structure is needed. The other sort of data management is that at plant level. At this level, the same data model should manage all modules of a plant. Which includes the composite data structure for each plant module, and the topology relations between plant modules. The logical structure at this level reflects the real plant structure, and is either a tree or a rooted graph.

Compared to the rooted tree in Bracketed tree OL-systems or Parametric L-systems, the FSP data model is fundamentally different. The rooted tree is the direct graphic result produced by L-systems, in which plant modules are represented by graphics. The FSP data model is a collective data management tool and supposed to be a part of the rewriting systems. The purpose to have an FSP data model is to allow effective access and modification of FSP data of each plant module. It represents plant modules by data elements (or graph nodes if the data model is a graph), and the adjacency between plant modules by relations between data elements (or edges if the data model is a graph).

Besides the logical structure of data elements for static descriptions, the basic definition of a FSP data model should include also data operations for dynamic descriptions and integrity constraints. For the data operations, functions for access and modification of data are indispensable. The integration constraints highly

depend on the specific technical environment but some are basic, such as the edges should have existing graph nodes as source/target nodes.

2.4.2.3 Data models in practice

A general data model for vector graphics, called **Scene Graph** [50, 69-71], is widely used in graphics editing applications. It allows the arrangement of logical (and spatial) representations of a graphical scene [72, 73] by vector-based geometry manipulation. Typically, a scene graph is a data model with a tree structure or a DAG (Directed Acyclic Graph). In a scene graph having a tree structure, a parent node has one or more child nodes, and nodes other than the root usually correspond to geometric objects, such as spline surfaces. In a scene graph, geometric objects can be iteratively grouped into **Layers** in linear or hierarchic manner. A linearly layered scene graph consists of shapes or groups of shapes at the same compositing level, while a hierarchically layered scene graph consists of nested shapes or groups of shapes. In a layered scene graph, some processes such as color-fading can be carried out individually on a layer without side effects to the others. In some FSP modeling platforms, the layer concept is equivalently used in the form of spatial scales to represent a plant at different levels of detail/spatial resolution.

The most significant advantage of the scene graph is the combination of logical and spatial (i.e. topology and geometry for FSPMs) modeling capabilities. In the scene graph, a logical modeling effect applied to a parent node propagates to every child node; an action on a collection of graph components is applied to the components automatically. Mostly, this process is realized by concatenating the geometrical transformations bound to each group. Through the combination of transformation propagation and graphics layering, vector graphics can be manipulated efficiently. There is a special advantage for L-systems using turtle commands. In fact, the turtle commands and the geometrical transformations in a scene graph are essentially both graphical operations on a relative or local coordinate system. Hence, the turtle commands of L-systems can be directly used

to represent the geometrical transformations in a scene graph, and the scene graph can be operated using turtle commands without additional grammatical settings. However, the scene graph has also some obvious shortcomings for FSP modeling. It does not distinguish different topological relationships between graph nodes, and has no means for plant functional property management as well. Besides, there is an essential topological disadvantage for using the scene graph as FSP data model. The transformations, equivalent to the turtle commands, are presented in the graph as topological nodes just like the shapes, but they do not represent any real world objects (i.e. plant organs). Therefore, it is clear that direct use of a scene graph is not appropriate and specific adaptions to suit the requirements of FSP modeling are needed.

On the other hand, another general data model for objects with various properties, called Property graph [74-77], is widely used in data management applications. It consists of a set of nodes (also called vertices) and directed edges (also called arcs). Within a property graph, each node or edge has a unique identifier and a set of properties of the form of key-value pairs. Besides, each node has a set of outgoing and incoming edges, and each edge relates to exactly two nodes with a fixed direction from a source to a target node. Particularly, when two nodes are connected by multiple edges at the same time, the property graph is a multigraph. When nodes or edges are tagged with labels, the property graph is called labeled property graph. The most important characteristic of the property graph is that the various data of a component or module of an object are regarded as the properties of the module. Essentially, different kinds of properties of the same module are organized into a set (i.e. indirectly related), and each set is distinguished as a node and identified by a unique id. The relations between nodes are regarded as edges, and can be distinguished by different edge types. The advantage of this data model is that it clearly defines the relations between both properties and nodes. The disadvantage is that it is too general to be directly used for a specific domain.

2.5 Synthesis of technologies and theories

Different comprehensive L-system based FSP modeling platforms have been formed by synthesizing the theories and technologies discussed in previous sections of this chapter in different ways. In this section, the different syntheses are introduced, and the resulting platforms, especially the research target platforms of this project (i.e. GroIMP and OpenAlea), are compared in details.

2.5.1 Synthesis of different platforms

The efforts to provide a specialized plant modeling tool started from the research of Lindenmayer, and continuously evolved by absorbing and synthesizing the theories and technologies which emerged during the same period. The whole process so far can be roughly divided into two stages.

The first stage includes the synthesis of research results and theory in the field of linguistics, including formal languages and Chomsky hierarchy, and in the field of computer science, including early third-generation programming languages [78] (3GL) (aka imperative languages) and turtle graphics. The focus on this stage is to support the plant structural modeling. In the 1990s, the parametric L-system based platforms (these early modeling tools were often referred to as programs because of their simplicity but we uniformly call them platforms) were the mainstream. Typical ones include cpfg [48, 79-81], grogra [56]. The modeling languages provided by these platforms directly use literals, or individually managed constants/variables of primitive types to allow FSP data to be manipulated by functional or structural simulators. The *Module* is defined as the basic syntactic construct and structural units correspond to a single or a set of plant organs. Only rule grammars are allowed for the development of simulators. Line segments are used to represent the plant modules. These syntheses provide plant graphic rewriting systems with limited modeling applications, such as computer animation.

The second stage includes improvements to the previous synthesis with emerged technologies in the field of computer science, including the property graph/scene graph based data models, the advanced third-generation programming languages (mostly the object oriented languages), and 3D graphics libraries. With these improvements, the platforms gain the capability for modeling sophisticated function - structure interactions within and between plant modules.

Precisely, in the early 2000s, the upgrade of cpfg, lfpg [82-84] was introduced with a special name called L+C given to its modeling language. It is one of the first systems to adopt the object-oriented technology and supports hybrid grammars combining L-system syntax borrowed from C++ and original C++ syntax. It does not provide a vector graphics library but four vector structures to support 3D lines representing plants. It also does not provide a FSP data model but an object database to facilitate the manual management of structural and functional data, which has been added as a shared tool to cpfg. By combining cpfg and lpfg with further shared tools like editors and 3D surfaces, this resulted in the platform L-studio [85, 86].

In 2008, GroIMP as the upgrade of grogra, including RGG (Relational Growth Grammar) and its Java implementation XL was introduced [58, 87-92]. It is one of the first L-system languages to adopt a FSP data model, namely the RGG graph. It is scene graph based property graph but with a more general structure, i.e. a rooted graph. For the RGG graph, specific syntax applying data operations to the graph and ensuring its integrity constraints are defined, and *modules* of types extended from the types defined in the IMP-3D library are used as nodes. Therefore, the nodes are Java objects that contain geometrical fields originally defined in graphical types (such as “length”) and functional fields defined through type extension (such as “absorbed light”). Edges are Java objects containing source and target nodes as data fields. Combining with tools like JEdit as code editor, code file explorer and 3D display, resulted in the platform GroIMP.

In 2012, a Python implementation L-py [93-96] with grammar largely borrowed from cpfg and lpfg was introduced as a supplement to the FSP modeling platform OpenAlea that was originally introduced in 2008 as a component based modeling platform [28, 97, 98]. It is one of the first L-system languages to adopt a dynamic programing language, and just like cpfg and lpfg, its technical basis is the parametric L-system. To make use of the existing FSP data model of OpenAlea, i.e. MTG (Multiscale Tree Graph), two-way conversion mechanisms between L-system strings and MTGs are provided, e.g. primitive *mtg2lstring (mtg, {parameters})* converts a "mtg" object with key-value paired parameters organized by a Python dictionary. The L-system string was extended from a bracketed string representing a single-scaled branching system to a string representing a multi-scaled branching system to enable this mechanism. To make use of the existing graphics library PlantGL and to control geometry in the specific language environment, adapted turtle commands are introduced, e.g., a generic primitive *@g (geometry)* is provided to allow the modeler to include any graphical objects of types defined in PlantGL.

2.5.2 Differences between the platforms

In general, the L-system based FSP modeling platforms can be roughly divided into two categories: graphical rewriting systems and graph rewriting systems. Cpfg, lpfg and grogra are graphical rewriting systems that combine string rewriting and turtle interpretation. Each derivation step includes two main sub steps, one is string rewriting, and another is turtle interpretation of the rewritten string. With graphical extensions, these systems can produce realistic plant structures. XL contains a rewriting formalism that generalizes string rewriting to graph rewriting. It provides RGG graphs as plant data models to algorithmically produce an updated graph from an original graph. The grammars that can be specified within the XL language are of a specific type of formal grammar, the graph grammars [99]. A typical graph grammar includes a set of graph-rewriting rules of the form $L \rightarrow R$, where L and R

are called pattern and replacement graph respectively. A graph-rewriting rule is applied to a host graph through two steps: 1. seeking for a match of the pattern graph in the host graph, 2. replacing the matching part by an instance of the replacement graph. L-py, on the other hand, is a special rewriting system, which is based on the combination of string rewriting and turtle interpretation similar to the graphical rewriting systems cpfg, lpfg and grogra. Although it provides grammars to allow the bidirectional translation between L-system strings and the data model MTG, the translation is not mandatory and automatic for each deviation step. Hence, it is essentially still a graphical rewriting system, but with an option to manually build a temporary graph rewriting system by the modeler on top of the platform.

In detail, the differences between L-system based FSP modeling platforms, particularly between GroIMP and OpenAlea lie in design and implementation of the platform components, i.e. L-systems based modeling languages, FSP data model, and the graphics libraries.

The grammar of XL, i.e. RGG, was introduced in 2008 [91] together with the RGG graph (and IMP-3D graphics library) with the purpose to link the graph with the grammar to form a graph rewriting system. The language XL thus includes the constructs not only for rewriting but also for data operations on the graph. The graphics types (and their extended types) defined in the graphics library are a part of the alphabet of the RGG. The grammar of L-py is based on the grammars of cpfg and lpfg, which are essentially string rewriting grammars. The early version of OpenAlea is a component-based platform allowing FSP modeling by visual programming [28]. It synthesizes the FSP data model MTG emerged before its introduction (in 1998 [100]), and the re-engineered graphics library PlantGL in Python emerged after its introduction (in 2009 [55]). The L-py was however introduced after this first synthesis, in 2012 [93]. The grammar does not include the

constructs for data operations on MTG but for translation of MTG and making use of PlantGL.

In terms of implementation of grammars, the RGG was implemented to XL in the Java programming language [101-103], for which variables need to be typed during coding and the program cannot be changed during execution. The implemented L-py is based on the Python programming language [104, 105], for which variables do not need to be typed during coding and the program can be changed during execution. XL has a rule-styled grammar while L-py has a statement-styled grammar.

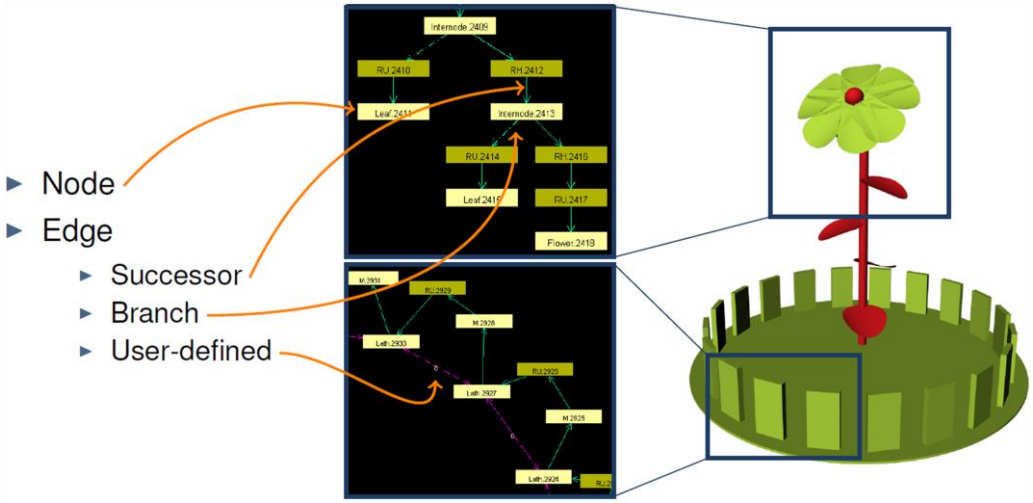


Figure 2.12 An example of single scaled RGG graph [13]

RGG graph has a rooted graph structure that can better suit the needs for description of a wide variety of plant structures than a tree structure. Its early version includes three basic types (successor, branch, refinement) of edges to allow the description of topological relationships between graph nodes, and arbitrary edge types defined by modelers are allowed for special modeling cases as well (c.f. Figure 2.12 [13]). All components of the RGG graph, including nodes of both shapes and transformations, directly correspond to the grammatical symbols of the

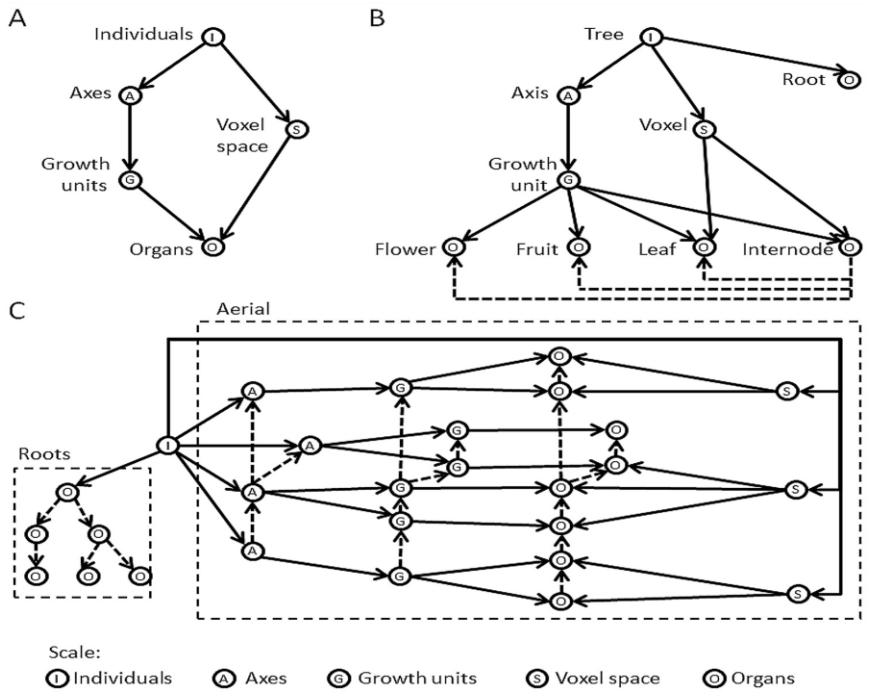
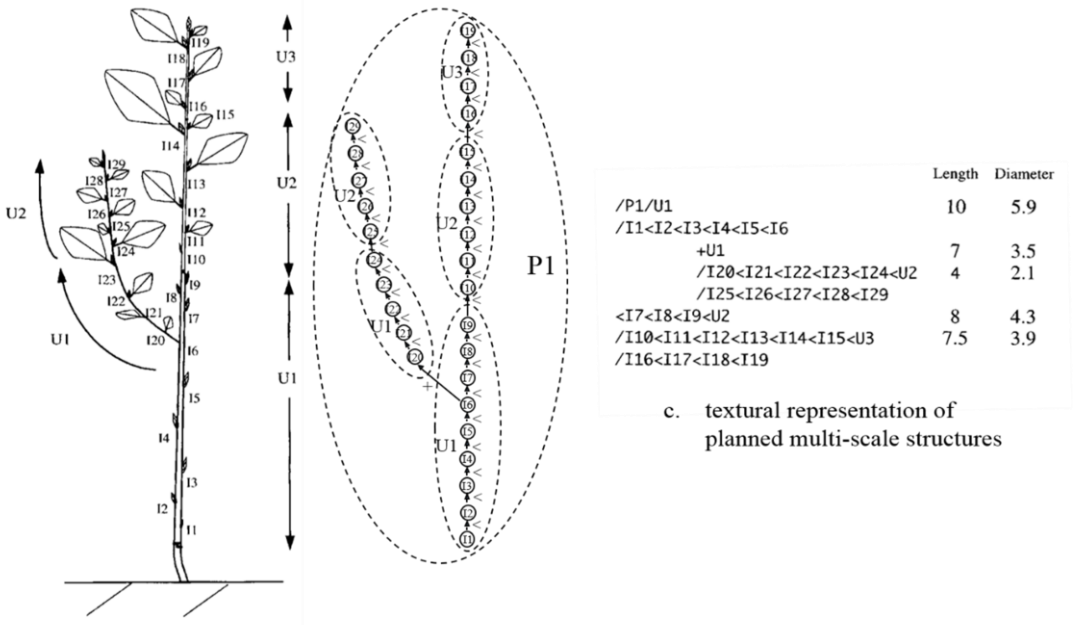


Figure 2.13 An example of three-part graph consisting of a scale graph (A), a type graph (B) and an instanced graph(C) [15]

string describing the graph grammar. The advantage of the direct correspondence is that it ensures that the modification of strings has intuitive and automatic effects on the graph, gives GroIMP high modeling interactivity, and essentially ensures the methodological evolution of string rewriting to graph rewriting. However, there is also a disadvantage of the setting. In fact, because the transformations appear as RGG graph nodes just like the shapes, there is no one-to-one relationship between the graph nodes and real plant modules, and thus the topology of the RGG graph does not match the topology of the real plant. Besides, this early version of the RGG graph only support static expression of plants at different scales, did not effectively consider dynamic modeling of plants at different scales. In 2014, the early version of the RGG graph was supplemented by introducing multiscale data structure components and data operations to the graph [15, 106]. Two sub graphs as metadata/schema were added to form a new version of RGG graph, namely three-

part-graph, which consist of a sub graph called *type graph* responsible for the description of types using at different scales, a sub graph called *structure-of-scales* (i.e. scale graph) responsible for the description of scale hierarchies, and the original graph as *instanced graph* (c.f. Figure 2.13 [15]). Meanwhile, the relevant grammatical updates have also been made to the XL modeling language accordingly, e.g. addition of multiscale grammatical symbols to the RGG alphabet.

The MTG was introduced in 1998 [100] as a method to digitally abstract and encode the architecture of real world plants (c.f. Figure 2.14 [12]). The design of the early version of MTG was focus on universality and intuitiveness. It has a multiscale tree structure without consideration of the shape types and transformation propagation. Every node in the MTG corresponds to a single or group of plant modules. The metadata of MTG [107] is included but not present as a part of the graph. The early version of MTG can only encode the plant skeleton



a. concretely planning of plant scales

b. schematic representation of planned multi-scale structures

c. textural representation of planned multi-scale structures

Figure 2.14 Encoding plant structure in MTG [12].

under global coordinates systems because it does not consider 3D graphics types [12, 100, 108]. The PlantGL library described in 2009 [55] makes MTG suitable to encode plants realistically with graphics object (c.f. Figure 2.15 [11] of types defined in this library.

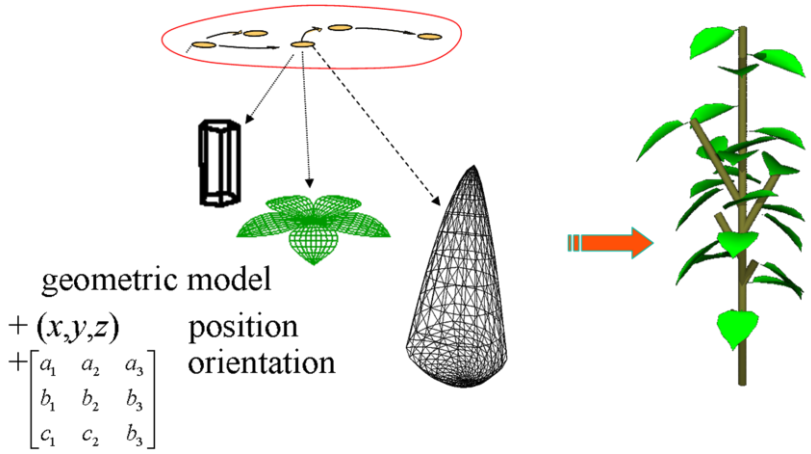


Figure 2.15 MTG with geometric models linked to each vertex [11]

In terms of implementation of graphs, the RGG graph is a scene graph based property graph, its primary nature is of scene graph. Which means it is of typical property graph, functional properties (i.e. key-value pairs) of a node cannot be directly added. In GroIMP, the RGG graph is implemented by a collection of related Java objects, in which nodes are Java objects of *module* types extending graphics types or turtle command types. The addition of functional properties is enabled by the type extension to modules. The MTG is a property graph based scene graph, its primary nature is that of a property graph. This means it is a typical property graph, functional properties of a node can be directly added. In OpenAlea, MTGs are implemented by a nested Python dictionary[105], in which nodes are entries of the nested Python dictionary. The functional properties can be added by applying MTG data operations that correspond the action of adding an entry of a Python dictionary. The graphic properties of a MTG node normally are not located in the MTG object,

but in a special object of Scene type available in PlantGL. So the graphics are not structured logically, thus the propagation of local transformation of a typical scene graph does not work. Consequently, every shape object is globally transformed from the coordinate origin.

The IMP-3D library of GroIMP is designed with all graphics types also graph node types. Some of them, i.e. the shaded shape types, have also turtled nature. That means when a shape is put into the RGG graph as a node, the current position of the turtle state will be changed from the start to the end location of the shape. Besides objects of the graphics types, RGG graph nodes consist of turtle commands as well. The alphabet of RGG includes both shaded shape types and turtle commands. The PlantGL library is designed with all graphics types not being the MTG node types but its property types, which do not have turtle nature. The alphabet of the L-py grammar does not include the graphics types but turtle commands, while the MTG does not include turtle commands but the objects of graphics types. The turtle commands are related to a local coordinate system while the graphics types are related to a global coordinate system. These are parts of the function of two-way conversion between L-system strings and MTGs, e.g., MAppleT map transforming turtle commands to the graphics types *Translated*, *Scaled*, and *Oriented*.

In terms of implementation of graphics libraries, the IMP-3D library [109] is implemented in Java as a plug-in of GroIMP and mainly defines 3D graphics types in the package *de.grogra.imp3d.objects*. This consist of types for RGG nodes that derive from the same root type *de.grogra.graph.impl.Node* in the RGG plug-in. Where all types (including *de.grogra.graph.impl.Edge*) for RGG graph constructs are defined. The turtle commands are defined as internal package *de.grogra.turtle* of the RGG plug-in, and all the command types are derived from the same root type *de.grogra.graph.impl.Node* as well. In this way, the objects of all 3D graphics and turtle command types or their child types are ensured to be RGG graph nodes.

Besides, all 3D graphics shapes derive from another root type *de.grogra.imp3d.objects.Transformation*, which is used to update the current position from the start to the end position of the shape. This ensures the turtled nature of the graphics shapes in the IMP-3D library. With such implementation, the application of graphical computing for a RGG graph node is a sequential process. Equations representing vector graphics types may have multiple equivalent forms containing different parameters. In the IMP-3D library, this phenomenon is enabled through the Java overloading mechanism [101-103], i.e. providing multiple overloaded constructors with different parameters. The re-engineered PlantGL library [110] was implemented in Python and is defined as an module independent from modules for MTG and L-py, it consist of types for the scene graph, but not for the MTG as there is no inheritance relationship similar to the IMP-3D library. In fact, it is technically impossible to have such inheritance in OpenAlea, as the MTG is not a graph of objects but a graph of abstract vertices and edges based on nested Python dictionary with numerical keys as id. Types of shapes and transformations are both defined in PlantGL as Python classes. Particularly, the transformations take the shapes as parameters of their constructors to generate the transformed shapes. Hence, they are derived from the same root type as the types of shapes, and with such implementation, the application of graphical computing for a MTG node is a recursive process. In PlantGL, the multiple equivalent forms of vector graphics equations are abstracted by creating a special function of the form *__init__(self, *args, **kwargs*) [105]. There, *__init__* is the function for initializing newly created instances by default, and **args* and ***kwargs* are Python syntax to define functions with an indefinite number of parameters.

Chapter 3

REQUIREMENT ANALYSIS AND TECHNOLOGY SURVEY

This chapter is mainly a survey of the existing technologies for the integration of different FSPMs based on requirement analysis. The background knowledge in the domain of software engineering, namely software reuse, integration and interoperability are introduced as the basis of the technologies. The requirement analysis is based on but not limited to the needs of the FSPM Apple project, the purpose to do so is to allow the design and implementation of the integrative interface to not only fulfill the requirements of our specific project, but also to provide a general solution for this type of problem. Here the general solution includes the technological basis of the integration and specific technologies designed on top of it. The former aspect is introduced in this chapter and the latter aspect is introduced in the next chapter. As the purpose of the PhD project is the construction of a complex FSPM reusing existing FSPMs, we therefore first give an overview about the software reuse and the integration of different software.

3.1 Complexity and requirement analysis of the integration

As introduced in the first chapter, this PhD project aims at integrating the two different FSPMs. The intuitive reason for doing this is to avoid duplication of work, but the root cause is the contradiction between limited resources and near-infinite modeling complexity. Actually, like all other kinds of models, FSPMs abstract and simplify only a finite range of plants to a finite extent due to various constraints, such as the limited available resources. It is practically not possible to model all physiological and environmental aspects of large complex botanical systems with many species by a single FSPM. To model complex botanical systems for a wide range of plants to a considerable extent, the capability of reusing the existing FSPMs on different platforms is desired. Which leads to the foundations of the domain of software engineering, and the background knowledge for software reuse, integration and interoperability.

3.1.1 Software reuse, integration and interoperability

The NATO Software engineering conference in 1968 gave birth to the field of software engineering [111, 112]. At this meeting, the so-called software crisis, namely the problem of building software that meets all requirements and guarantees quality in all aspects (operation, modification, transfer) in a manageable manner, was first introduced and discussed as the core topic of the conference, and software reuse was first proposed as a way to overcome the crisis. The software reuse is about the process of using existing software artifacts to build new software rather than building them from scratch. The reason it was supposed to be a potentially powerful way of improving software practice and providing a solution for the software crisis is that the time and effort required for building software systems can be obviously reduced by the reuse of existing software.

In the following two decades after the conference, many research and practice activities on software reuse technology were carried out. However, due to various technical and non-technical factors, the software reuse has not been widely accepted as a standard practice. In 1992, C.W. Krueger [113] introduced four dimensions that software reuse technologies might involve, i.e. abstraction, selection, specialization and integration, and analyzed the reason why the reuse is difficult by following the dimensions. He found the primary requirement for implementing a software reuse technology is to provide natural, succinct, high-level abstractions that describe artifacts in terms of "what" they do rather than "how" they do it. As the essential dimension, the abstraction of software artifacts is however very complicated, especially when the artifacts are large and complex ones.

In the 1990s, the maturity of object-oriented methods and technologies provided powerful technical support for software reuse [112, 114]. In particular, the development of software component technology has injected new vitality into software reuse, making its research a hot spot again. It is regarded as a realistic and feasible way to solve the software crisis and improve production efficiency and quality of software. At the same time, it has become a solution to avoid duplication of labor in software development, and to some extent reduce the cost of software development. The reuse practice based on component technology mainly includes two types, software composition and software integration [115].

Software composition refers to the process of seamlessly connecting different software based on a certain software component model, following a specific software architecture, through a standard interface mechanism, and assembling into a new software system or software component with certain functional characteristics. Software composition solved the software crisis to a certain extent. It is suitable for assembly of components in homogeneous environments, especially in stand-alone systems. From an application perspective, it is mainly a means to solve the problem of efficient development and upgrade of software systems. It is

the composition of components designed for reuse purposes, which enables a software reuse in a complete sense. That is, software (component) based application system construction (development with reuse).

In 1990s, with the development of computer networks, especially the popularity and application of the Internet, the problem of software reuse in a distributed and heterogeneous environment emerged. To address the problem, the concept of software integration [116-118] was introduced. Software integration (also called software system integration) refers to the process of homogeneously or heterogeneously connecting and coordinating different existing software (software components, non-software components or legacy systems) in a distributed environment, based on a certain architecture, following a specific software architecture, through a specific infrastructure (integration middleware [119, 120]), and meeting certain performance requirements (such as real-time and security). It is suitable for the assembly of components in distributed heterogeneous environments. From an application perspective, it is mainly a means to solve the problem of communication and interoperability between legacy systems or island systems. In the software integration process, in many cases, some non-componentized legacy systems are involved. Due to historical reasons, these systems were not designed for reuse purposes at the beginning of the development. Therefore, in the process of software integration, there is a process of re-engineering, encapsulating, and building components of legacy systems.

In the development of software engineering, various aspects of software integration have been discovered in practice. Typical ones include the integration for two layers of the widespread three-tier architecture, i.e. data/information integration, the process/business integration [116, 117, 121]. The former is about passing information back and forth between different software systems, and assuring that the information is understood by these systems to produce useful results. The latter is about coordinating processes or workflows between different

software systems. That means a workflow can start in system A and continue in system B, and the flow is meaningful and in line with expectations. Diverse methods and technologies [118, 122, 123] for the software integration have been introduced from different perspectives and application fields.

As one of the practice types of software reuse, the technology used in software integration generally consists of three categories. The main category is undoubtedly component technology. In addition, technologies that respond to complex situations in a distributed and heterogeneous environment are also needed. Which mainly include two categories, i.e. middleware technology (including communication technology, distributed object computing technology), and software architecture technology.

Precisely, the component technology refers to the use of software components for software development. Unlike the process-oriented technology that allows the reuse of functions and object-oriented technology that allows the reuse of classes, component technology allows the reuse of program modules with full specific functionality. Each component provides some interfaces to expose its functions, through which components from different sources may be assembled to rapidly build a large application that meets all requirements and guarantees quality in all aspects (and at a relatively low price). The main characteristics of "components" are different from ordinary software, such as reusability (common/general), customizability (setting parameters and attributes), self-containability (relatively independent with relatively complete functions) and interoperability (multiple components work together). Process-oriented and object-oriented technology typically generates two types of software: application-specific executables and API libraries for general-purpose software development. The former contains various special specific functions that are required, but must be created from start to finish, many of which are low-level repetitive work; the latter, although generic, does not meet the specific needs of specific applications. Component technology provides a

third way to combine the reusability of libraries with the customizability of specific programs, allowing users to customize their own specific applications with reusable components. Therefore, a component is similar to an “executable program” in some respects and a “library” in other respects. Essentially, a reusable software component is an independent executable unit defined by its interfaces that can be included directly in a software system or referenced as an external service. By parameterizing operations through interfaces of components, their functionalities are available for interaction. A component is either a software element or an external service. In the former case, it has two related interfaces, namely the ‘requires’ (or ‘required’) and ‘provides’ (or ‘provided’) interfaces (c.f. Figure 3.1 [1]) which reflect the functionalities needed by them and supplied to others. In the latter case, it has only the ‘requires’ interface. A software component conforms to a component model, which defines the architecture of the component and how to manipulate it and interact with the other elements.

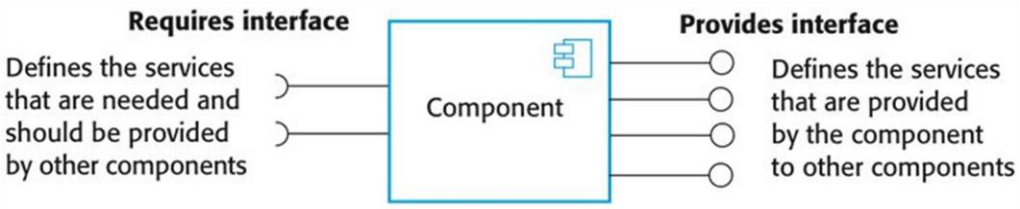


Figure 3.1 Component interfaces [1]

Middleware in general is a service system between the application system and the operation system. (c.f. Figure 3.2 [5]). It provides standard programming interfaces and protocols to allow interoperability on different hardware and operating systems, and can also dynamically respond to some operational performance requirements, such as real-time, security. The introduction of middleware technology in software integration makes it easy to integrate existing

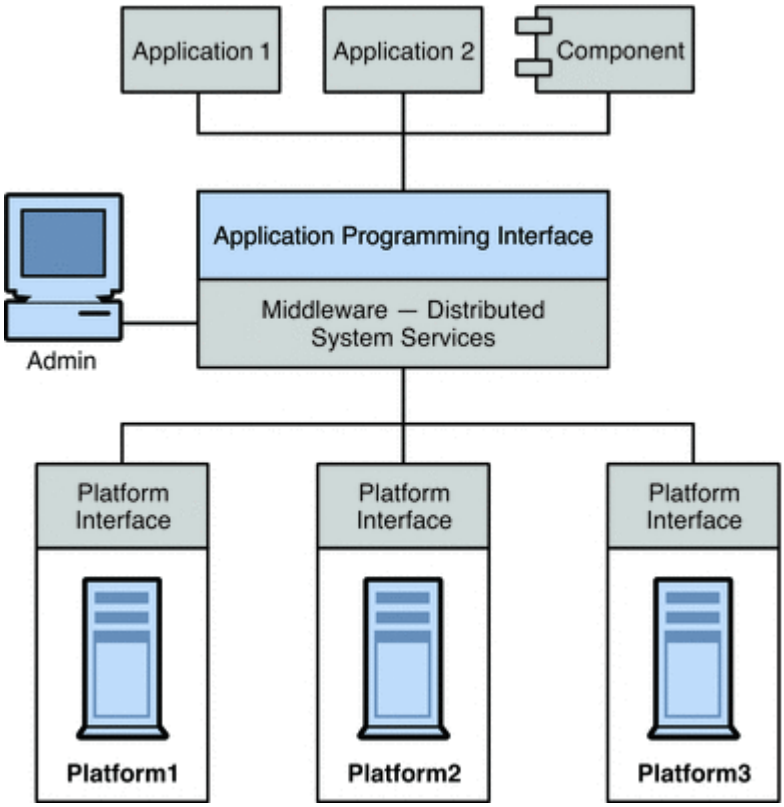


Figure 3.2 Middleware architecture [5]

software systems and realize the integration of data, service and presentation layers. It also reflects the openness and scalability of software system development. When using middleware, it is often a set of middleware integrated to form a platform (including development platform and running platform), but in this set of middleware there must be a communication middleware, namely middleware consisting of platform and communication. This definition limits the use of middleware only in distributed systems, and distinguishes it from supporting software and utility software. It can achieve access transparency and location transparency of resources to distributed software systems, and ensure interoperability between objects in distributed homogeneous or heterogeneous environments. Developers are thus able to access and integrate a large number of software resources, regardless of the tools or languages used by their developers or

development processes. In a narrow sense, middleware is a system that allows independently developed software that operate on different network platforms to cooperate with each other. It hides some complexities of building a distributed software and allows developers to focus on issues at the application level rather than low level.

In software integration, software architecture is an integrated architecture, which is the guiding basis for developers of distributed software application systems to integrate software. A software architecture is the structure of a program/system component, the relationships between them, and the principles of design and evolution over time. The following mechanisms and methods are commonly used to describe software architecture: architecture description languages, architecture viewpoints, architecture frameworks and architectural patterns. An architecture framework captures the "conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders" (ISO/IEC/IEEE 42010). An architectural pattern, or design pattern, is a general, reusable solution to a commonly occurring problem in software architecture within a given context. In practice, specific design patterns have been introduced to enable the EAI (Enterprise Application Integration).

Since the end of the last century, software interoperability has received attention and is being studied as a major method of achieving software integration [124]. Various architectures enabling software interoperability have been introduced. In different application areas and practical situations, software interoperability is endowed with different connotations not only at application level but also at the standard level [125-127]. Here is an example at the application level: in the application field of electronic government (eGoverment), the European Interoperability Framework (EIF) v1.0 [21] under the Interoperable Delivery of European eGovernment Services to public Administrations, Businesses and Citizens program (IDABC) was published in 2004 and defined Interoperability as

"the ability of information and communication technology (ICT) systems and of the business processes they support to exchange data and to enable the sharing of information and knowledge". While in the area of health care, the Office of the National Coordinator for Health IT (ONC) of the USA defined it in the Shared Nationwide Interoperability Roadmap version 1.0 [128] as "the ability of a system to exchange electronic health information with and use electronic health information from other systems without special effort on the part of the user". Here is an example at the standard level: the ISO/IEC 2382-01:1993 [129] described interoperability as "The capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units". While the IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries [130] defined the interoperability as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged." Even the same standardization organization gives different standards for the interoperability at the same application field, e.g. K. Kosanke [126] compared two ISO standards for software interoperability (ISO 15745 and ISO 16100), with their focus on interoperability within manufacturing applications and between manufacturing software units, respectively. These definitions are ambiguous and misleading, some are not even comprehensive, e.g. the definition of interoperability in ISO/IEC 2382-01:1993 focused on the technical side and did not consider organizational issues such as the user of a program to be another program. Moreover, practical aspects of software integration and software interoperability are very much overlapping, which has led to conflicts and contradictions. Some people see interoperability as the result of integration [131, 132], while some see it the other way around [133]. Others [134, 135] argue that an integrated solution not only allows the subsystems to talk to each other in their current state, but also provides backward and forward compatibility with future versions of each other. Nevertheless, interoperability reflects only an immediate form of functionality between different subsystems, future upgrades or developments or improvements

to any of the subsystems can cause interoperability to cease. In brief, the two concepts are different, and there is no causal relationship between them.

From the perspective of software reuse, the integration of different FSPMs refers to the means that enable the existing functional and structural data processing programs to act as one program so that a specific simulation purpose can be achieved without developing a new complex FSPM that may duplicate the development work of existing FSPMs. It is obvious that the interoperability between the FSPMs is the key to the success of the integration. Moreover, FSPMs are normally designed without consideration of technical upgrades, while the modeling platforms have potential technical upgrades, developments or improvements and provide certain downward compatibility just like other software development tools. The integrated FSPMs can work together for a relatively long time thanks to such technical stability at model level and downward compatibility at platform level. Therefore, in this project, we regard software interoperability as the technical basis to enable software integration, not the other way around.

3.1.2 The target FSPMs of the project: overview

In this project, the target FSPMs to be integrated are MAppleT and a GroIMP based transport or radiation model. MAppleT [26] is a graphical rewriting based functional and structural model simulating the growth of apple trees on a stochastic basis and taking the effects of gravity on branch shape into account. The GroIMP transport model [136] is a GroIMP based functional and structural model simulating the water and sugar transport in an apple branch. The GroIMP radiation model [91] is a GroIMP based functional and structural model simulating the light absorbed by an object located in a scene. The goal of the integration is to provide a complex FSPM without duplicative modeling work. The integration here mainly refers to the process to enable the interaction between apple tree growth and physiological

function. Water and sugar transport is our main focus, with light interception as one of the factors influencing stomatal conductance and thus water flow.

MAppleT models the plant topology and geometry by a mixed approach that combines stochastics and biomechanics. A hierarchical hidden Markov model is used to model the development of growth units along both axes and branches, and a biomechanical model is applied to compute the stem at metamer scale considering the primary and secondary dynamics and fruit growth within a year. In [26], at a time interval of one simulated year, the architecture of an apple tree generated by L-studio is represented as MTG to quantitatively compare the virtual tree with a real tree. For the creation of organ geometry, the library PlantGL [55] was applied.

The scale hierarchy of an apple tree modeled by MAppleT includes the tree, axes, GUs (growth units), and metamers with topological connection types including succession, branching and decomposition.

The GroIMP transport model abstracts the dynamics of two related processes in an apple tree branch, i.e. water transport in the xylem from root to leaf and sugar transport in the phloem from leaf to all organs. The former is the basis of transpiration and the latter is the basis of carbon allocation. This model takes the branch structure of the apple tree in the form of an RGG graph, and computes the flux within the branch, i.e. the dynamics of the amount of flux passing each shape node. The computation is applied to a sequence of RGG graph nodes of shaded shape type that belong to the same branch. The physical principle followed in this model is Darcy's law [136]. The GroIMP radiation model computes the light amount absorbed by an object placed in a scene with customizable light sources. This model uses the technology of path tracing for radiation transport, i.e. light rays emits from a light source are traced to calculate a scattered ray by applying optical principles., and the Monte Carlo method for tracing diffuse reflection or transmission, i.e. the new direction is (pseudo-) random. The geometry of the scene, the optical properties of the objects and light sources are set as parameters to

compute the light amount intercepted or absorbed by any object. The computation of the radiation model thus is applied to each RGG graph node of shaded shape type.

On one hand, the GroIMP based FSPMs can take both single-scaled and multi-scaled RGG graphs, with topological connection types including one, two or three types out of the succession, branching and decomposition. On the other hand, it is logical that the GroIMP functional models take the role of servers that receive the modeling interaction requirements from MAppleT, because one plant can have multiple functions but only one growth algorithm. Thus, the interaction between MAppleT and the GroIMP FSPMs acts in a way that FSP data generated by MAppleT is reproduced in an RGG graph with original topology and geometry to allow the functional computing by the GroIMP models. Obviously, there are syntactic and semantic gaps which need to be bridged, e.g., the FSP data generated by MAppleT does not contain data fields for the flux or light, which are needed for GroIMP FSPMs to perform functional computing.

3.1.3 Requirements to achieve the project goal

From the perspective of software engineering, this PhD project is about the reuse of two existing FSPMs and the construction of a complex FSPM. On one hand, almost no FSPM that is expected to be reused was originally designed for reuse purposes. On the other hand, FSPMs that are expected to be reused are mostly based on heterogeneous technology environments (e.g. different programming languages, different platforms) with different aspects of plant abstraction. The reason is that the same research team often uses the same technical environment to do the same kind of research, and the reuse of research results of different research teams on different research directions will enable a more comprehensive understanding of the research object and will thus be more valuable. Consequently, not software

composition but software integration is a major kind of software reuse practice for FSPMs, and is the focus of this thesis.

The essential mechanism that executes different FSPMs as one program is obviously the cooperative processing of the functional and structural information of the same virtual plants. An information exchange between FSPMs is thus necessary. In order to ensure the information processed by different models are indeed for the same virtual plants, the exchange of the FSP information between different FSPMs is indispensable. It is noteworthy that when information is decoupled from the modeling environments, e.g. XL/L-py, IMP-3D/PlantGL, it will become data, which is the actual form of exchange. Only when the exchanged data are recoupled to another modeling environment, the FSP data become information again. Another notable feature is that the target FSPMs of an integration typically include a plant structural model and zero or more functional models. In biology, the basic assumption is that a structure is the basis of its functions; functions of a structure determine the performance of the structure. Consequently, a single structural model takes the role of "client" and the multiple functional models take the role of "servers".

In detail, the integration involves every aspect of a FSPM. The first is the *modeling platform – model* aspect. Although both modeling platform and model are software programs, they play rather different roles for a FSPM. The modeling platform provides the technical basis of the FSPM. At platform level, plant information produced by FSPMs based on the same modeling platform shares the same syntax and semantics. Hence, these FSPMs can use the same platform-level integrating infrastructure, i.e., processes for the platform-level interoperability of information. At model level, both information and simulator of a particular model have their own specific syntax and semantics, hence every FSPM has its unique model-level integrating infrastructure, i.e., processes for model-level interoperability of information and for synergy of simulators. The second is the

syntax - *semantics* aspect. Information involved in FSPMs includes plant and environmental information. Both consist of data organized in syntactic structures with given semantics. Data need to be exchanged with relevant semantics to enable the simulation by the receiving FSPMs. The third is the *dependent* - *independent* aspect. Because plant elements are biologically dependent on and interact with each other, FSPMs compute the FSP data of one plant element by taking into account inputs from one or more other plant elements. The plant information produced over one simulation step thus needs to be exchanged as a complete piece with consistent semantics in different syntax. In contrast, the different environmental information is normally considered independent from each other; hence, it does not need to be exchanged as a complete piece. The fourth is the spatial-temporal aspect. An execution of FSPM normally simulates particular plant functions at one or more structural extents (i.e. spatial resolutions/scales) with a specific simulation step length (i.e. temporal resolutions/scales). The fifth is the *topology* - *geometry* aspect. Being a part of plant information, structural information includes topology and geometry. In detail, topology denotes the adjacency relationships between a plant module and its neighbors, whilst geometry denotes the location and orientation of a shape presenting a plant module, which can be expressed by the geometric transformation between the plant modules and its parents (local transformation) or the root (global transformation). The sixth is the *internal* - *external* aspect. The FSP data in plant information captures properties of the plant itself, i.e., internal data. In contrast, the data in environmental information is external. As FSPMs focus on small spatial scale modeling, the evolution of internal data is frequently assumed to have no feedback on the external data, and the same external data is applicable for all involved virtual plants. The seventh is the *non-retroactive* - *retroactive* aspect. A non-retroactive integration denotes a situation in which the target FSPMs do not send the updated plant information back to the source FSPM. In contrast, a retroactive integration describes the case where the target FSPMs send the updated plant information back to the source FSPM and let the source FSPM take into account data on updated properties when computing new plant information.

Before FSPMs can be integrated, some preparations need to be carried out. One preparation is for plant properties. Similar to databases where different data fields characterize different properties of an object, different FSPMs originally organize data characterizing plant property information in different data field sets. However, the simulation of integrated FSPMs requires plant information with data fields from both source and target FSPMs. As the original plant information from the source FSPM does not match data fields needed by the target FSPM, hence, the data fields defined in the target FSPMs need to be added to the data field set of the source FSPMs and data fields need to be adjusted to the types available on platform where the target FSPM is based. The other preparation is for simulators. Originally, simulators of an FSPM update plant information by computing new data of a data field using old data of relevant data fields defined in the FSPM itself. However, to compute new data of data fields defined in the source FSPMs in case of retroactive integration, the computation also takes data from data fields defined in the target FSPMs as inputs. Hence, simulators of the source FSPM need to be adjusted. On the other hand, the types used to graphically represent plant modules in simulators of the source FSPMs might be different from those of the target FSPMs, besides of the structure alignment mentioned in the geometry-topology aspect at the data model level, adjustments of the simulator of the target FSPM at biology level are also needed. (e.g. a source FSPM uses a *Parallelogram* type available in its library for a leaf, a target FSPM uses a *Triangle* type for a leaf available in its library, then a leaf object produced by the source FSPM needs to be represented by, let's say two triangle objects which will be incorrectly recognized by the simulator of the target FSPM as two leaves. To have a correct biological interpretation, the simulator of the target FSPM needs to use an array of two objects of the *Triangle* type available in its library to represent a leaf, and the production rule needs to be changed accordingly.)

In order to have a standardized technology, we must consider the requirements of diverse projects, namely the integration of a various number of FSPMs. Here,

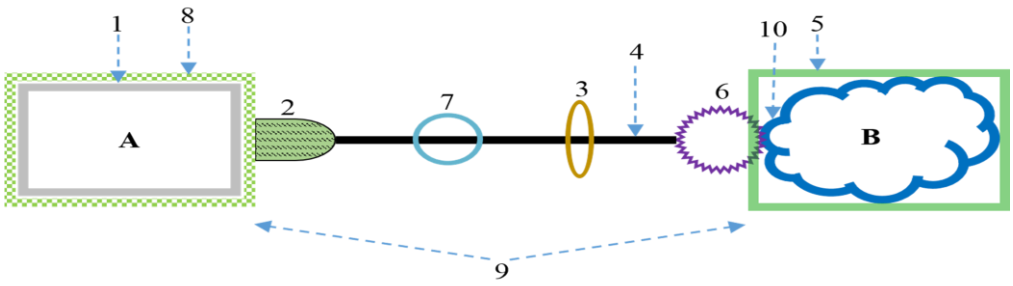
the integration case of the FSPM Apple project is a special case where only two FSPMs are involved. When there are more than two FSPMs to be integrated, the coordination of the execution of different FSPMs becomes necessary. The coordination has to conform to specific knowledge or settings of experiments, thus the participation of domain experts is indispensable.

To meet all the requirement aspects and achieve the integration of the two specific FSPMs, specific middleware has to be developed to enable the required interoperability. The middleware needs to be modularized as reusable components and loosely coupled with the FSPMs through “provided” or “required” interfaces to keep the independence of the FSPMs. However, the middleware obtained can only be used under the specific conditions of our project. To provide technical support for all the cases of integration, a set of technologies to support the integration of different FSPMs is better to be provided first, and then a middleware that fulfills the specific requirements in our integration case can be developed by applying these technologies.

3.2 Technology survey for the integration of different FSPMs

Being a particular kind of software, the technology categories [115] to achieve software integration, i.e. software component technology, middleware technology, software architecture, are logically applicable for the integration of different FSPMs. However, the existing technologies of each category are generalized standards that do not fully meet the obtained requirements, e.g. many of them are only applicable for specific languages or platforms. Therefore, the pragmatic approach is to establish a specific solution with full adaptability to the needs of the project based on a survey of different existing technologies. In this section, we firstly survey the existing technologies of each category for software integration,

and then we introduce some of them in detail as the conceptual basis for construction of our specific solution of FSPM integration.



1. Change A's form to B's form, which is about to a complete rewrite of A or B using standard architecture-specific frameworks
2. Publish abstraction of A's form
3. Transform on the fly, through data filters, mediators, scripts and other externally-imposed controls
4. Negotiate common form
5. Make B multilingual, which is about to make the subsystems capable of interacting in different forms
6. Provide import/export converters
7. Introduce intermediate form
8. Use wrapper
9. Parallel consistent versions
10. Separate B's essence from its packaging

Figure 3.3 Approaches for software interoperability [10]

3.2.1 Technologies for software integration: overview

David S. Rosenblum has summarized ten different approaches to achieving interoperability (abbreviated as D.S.R ten IOP approaches, c.f. Figure 3.3 [10]) with "form" referring to the representation, communication, packaging semantics from the perspective of methodology. These approaches mostly embody the roles of different integration technologies and the logical relationship between them, and can provide guidance on how to apply specific integration technologies.

In the category of component technology, the existing concrete ones mainly are technologies supporting the reuse of a component by creating a copy within the new software system. The typical examples include Common Object Request Broker Architecture (CORBA), Java Bean & Enterprise Java Bean (EJB), and Component Object Model (COM) & .NET. Through the comparative Table 3.1 [16], Philip T. Cox and Baoming Song summarized the characteristics of these technologies, and found that despite the differences in many aspects, such as the parameterization mechanism, these technologies have some basic features in common [16], which include the standards provided for the definition of the interfaces required for component communication and the message exchange mechanism for the interoperability between components.

	JavaBeans	COM	CORBA
Component	Module containing multiple classes	Module containing multiple classes or other implementation	Module containing any implementation
Interface	Java language	OLE IDL, which defines interfaces as collection of functions	OMG IDL
Connection	Via event and listener.	Via interface pointers	Via Interface Definition Language
Variability mechanism	Inheritance and aggregation	Genericity, containment and aggregation	Inheritance and aggregation
Platform	Multiple platforms	Windows	Multiple platforms
Implementation Language	Java	Any languages, but primarily use C++ and Visual Basic	Any languages
Distribution Mechanism	EJB, Internet, RMI (remote method invocation)	DCOM, Internet	An ORB
Self-description	Support via introspection	No	No

Table 3.1 Comparison of JavaBeans, COM and CORBA [16].

Ian Sommerville found that the situation of the multiple standards has caused difficulties for components developed using different approaches to work together [1], e.g., components developed for .NET and J2EE cannot interoperate. Consequently, the component based software engineering and software reuse is greatly hindered. Besides, these component technologies are highly complex with

steep learning curve. He suggested that component technologies adopt a service-oriented concept to address these issues. This means to establish standards supporting the reuse of a component by referring to it as a standalone service that is external to the software that uses it. The most important service-oriented component technology is the Web Services technology. As a common means for cross language/platform software interoperability, the Web Services technology combines a collection of recognized standards to define software components as web services over networks, and to allow their communication through XML messages. These standards usually include Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP) and Hyper Text Transfer Protocol (HTTP) respectively.

In the category of middleware technology, the existing concrete solutions mainly are different models/protocols allowing communication between distributed software applications or components, including RPC (Remote Procedure Call), MOM (Message-Oriented Middleware) and ORB (Object Request Broker). The RPC is a protocol for communication between processes. It allows a program to call procedures in a different address space (usually on another machine in the network) without explicitly describing the details of the remote call by the developer. That is, whether the procedure calls locally or remotely, the calling code is essentially the same. The MOM allows distributed applications to communicate and exchange data through messaging mechanisms. The messages are asynchronously stored, forwarded, transformed and routed. The OBR enables an application's objects to be distributed and shared across heterogeneous networks, i.e. interoperability between objects. All these middleware models make it possible for one software component to affect the behavior of another component over a network. The difference is that systems built upon ORB- or RPC-based middleware have components that are tightly coupled, whereas systems built upon MOM-based middleware have components that are loosely coupled. In an ORB- or RPC-based system, communication between components is straightforward and synchronous. That is

to say, there is no forwarding intermediary, and the caller must wait for the reply from the callee before proceeding to the next step. In a MOM-based system, a message is sent from a source application to a destination application through a messaging provider that mediates the messaging operation. This means that the provider manages the message by routing and delivering it. The source application can continue for further work once it has sent the message, confident that the provider maintains the message until a destination application receives it. [5].

In the category of software architecture of integration, the existing concrete software integration architectures mainly are embodied by different architecture frameworks and design patterns. As introduced previously, the interoperability enabled software integration is the state-of-art approach, we thus focus on the IOP architecture frameworks. One of the early IOP architecture frameworks is the four levels or aspects of software interoperability, including physical, data type, specification and semantic, introduced in 1997 [124]. Issues for interoperability at each of the four levels have been studied, e.g. procedure call versus message passing, systems interface definition, execution intermediaries and data type compatibility. In the following twenty years, through a large number of practices in specific application fields, different layered frameworks combined with technical solutions adapted to the application fields have been introduced.

In the field of eGovernment, which is about providing public services to people electronically and which involves interactions between heterogeneous roles ranging from different people to countries, the EIF version 1.0 [21] and version 2.0 draft [3] was introduced in 2004 and 2009 respectively. It is an IOP architecture framework for enabling the integration of the eGovernment Services of the member countries of the EU. The version 1.0 introduces a framework consisting of technical, semantic and organizational aspects. Despite the suggestion [9] to confine the organizational interoperability to standards or concepts handling the linkage of business processes and rename it to business interoperability, the version 2.0 draft kept the

organizational layer and added an additional legal layer with a political context applicable to all four layers (c.f. Figure 3.4 [3]). An architecture framework similar to EIF version 1.0, i.e. the European Public Administration Network (EPAN), adds a layer corresponding to contact/support of a structured customer and introduces the aspect of governance as a cross-cutting issue as an addition to the four layers (EPAN 2004). In the white paper "Standards for Business", the European Telecommunications Standardization Institute (ETSI) introduces a new layer between the layers for technical and semantic IOP, which corresponds to syntactic IOP (ETSI 2006).

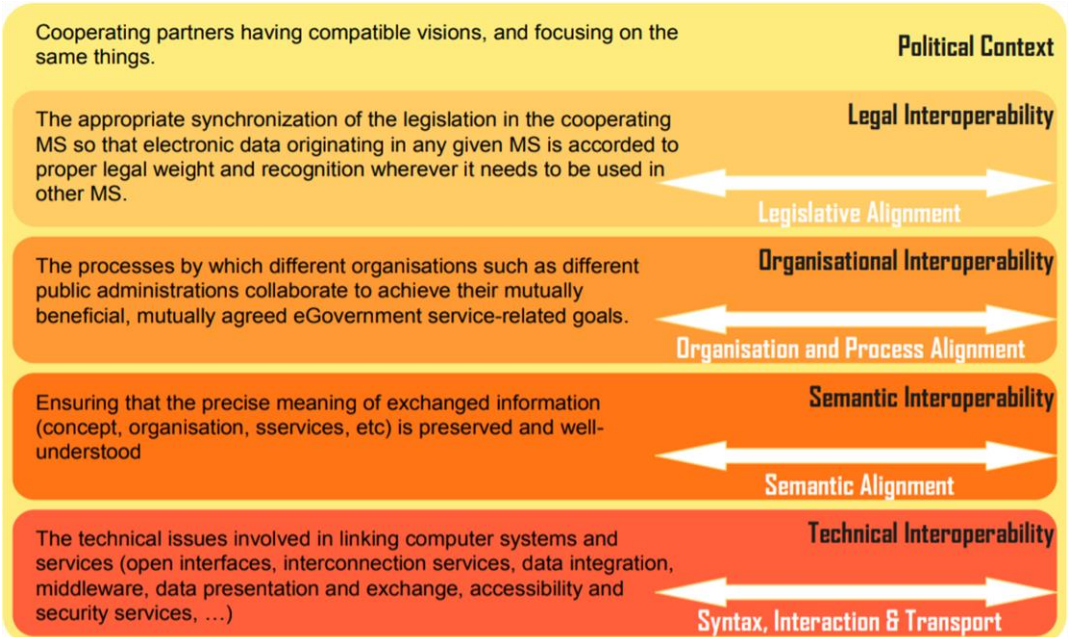


Figure 3.4 The interoperability framework of EIF version 2.0 draft [3]

Like the situation for the definition of interoperability, different application fields will give different interoperability frameworks. Even in the same application area, the frameworks given by different institutions are not the same. Besides, the frameworks change over time. As previously introduced, these conflicting and inconsistent taxonomies have been observed and discussed, but no universal

solution has been provided. In fact, it is understandable that it is not possible to provide such a static and universal framework to enable software interoperability because of the high diversity and variability of software development technology. Thus, it is pragmatic to refer to the existing standards and develop a realistic framework based on our project needs.

Layer of IOP	**Aim**	**Objects**	**Solutions**	**State of knowledge**
Technical IOP	Technically secure data transfer	Signals	Protocols of data transfer	Fully developed
Syntactic IOP	Processing of received data	data	Standardized data exchange formats, e.g. XML	Fully developed
Semantic IOP	Processing and interpretation of received data	Information	Common directories, data keys, ontologies	Theoretically developed, but practical implementation problems
Organizational IOP	Automatic linkage of processes among different systems	Processes (workflow)	Architectural models, standardized process elements (e.g. SOA with WSDL, BPML)	Conceptual clarity still lacking, vague concepts with large scope of interpretation

Table 3.2 Four levels of interoperability (IOP) [9].

It is easy to see that all the interoperability frameworks include explicitly or implicitly the syntactic and semantic interoperability, which is about to ensure the information exchange and consistency in understanding the exchanged information. From the perspective of layered interoperability frameworks, each layer needs to have a clear technical solution when a specific interoperability task is faced. The four levels of interoperability of H. Kubicek and R. Cimander [9] (abbreviated as K.C. IOP framework, c.f. Table 3.2 [9]), show the solution (and some candidate

technologies) for each layer, and the state of knowledge for the solutions. The fact is that the technical and syntactic interoperability has mature candidate technologies such as TCP/IP or XML, JSON for the solution, but there are only unstandardized concepts and methods available for the semantic interoperability, and there is even a lack of conceptual clarity for organizational interoperability (i.e. the automatic linkage of processes among different subsystems).

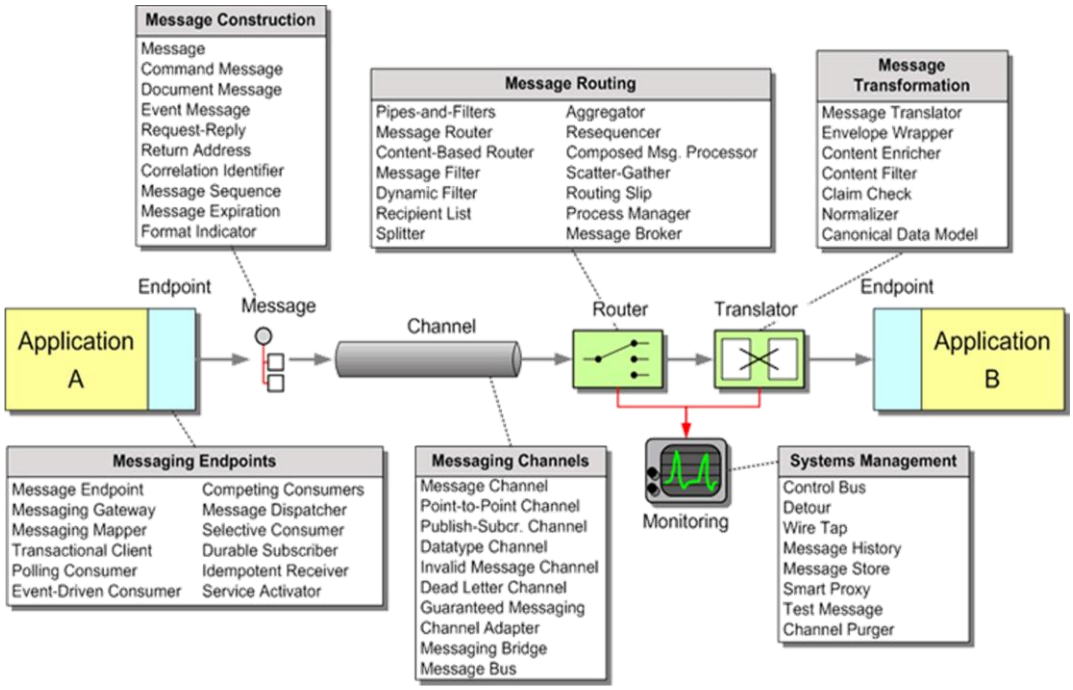


Figure 3.5 Design patterns for enterprise application integration [7]

The design patterns, as one of the two major technologies in the category of software architecture, are intensively discussed based on the integration practice of enterprise applications. An icon-based pattern language consisting of sixty-five integration patterns structured into nine categories was introduced to ease the difficulties of the software integration, especially the high complexity of Enterprise Application Integration (EAI) [7], which is about to facilitate the integration of

software applications (and hardware) systems across different enterprises. The patterns were collected from a large number of integration practices and provide technology-independent design guidance for developers and architects to describe and develop robust integration. This coherent collection of relevant and proven patterns forms an icon-based integration pattern language which can be used to describe the pattern. The EAI is enabled by different messaging EIPs including ten Channel patterns, fourteen Router patterns, and seven Transformation patterns. The additional Endpoint patterns, Message Construction Patterns, System Management Patterns (c.f. Figure 3.5) are also included to describe the produce-consume, monitoring, pack-unpack of messages. Following the EIPs, the enterprise application is essentially integrated through message-oriented middleware.

While discussing software integration, we must also note that the aim of software integration is to construct complex software systems reusing existing software. As the software is mainly about to process data/information, the integration of different software will surely bring problems of data/information interoperability, especially when complex data models are involved. This can be confirmed from the existence of syntactic and semantic IOP layers in most IOP architecture frameworks. The formal definition of data integration is to logically or physically combine data from different sources in different formats or data models into meaningful and valuable information and providing users/processes with a unified view of them. The method of data integration bridges the gaps of heterogeneous data formats/models, which logically has the potential to enable data to be processed collaboratively by heterogeneous software, i.e. the data integration methods enable the syntactic or semantic interoperability. Traditionally, data integration can be divided into two broad categories of methods, namely data warehouses and federated databases. Database warehousing technology physically integrates and stores data distributed across different data sources into a single data model and central database so a single user query can retrieve data from different sources. Typically, a data warehouse is established through a data pipeline to extract, transform, load (ETL)

data in heterogeneous data models into a single data model so data from different sources become compatible. Federated databases logically integrate data only by translating user queries into data source queries, i.e. decompose the user query into subqueries for submission to the relevant constituent databases and compose the result sets of the subqueries. To deal with the heterogeneous query languages of the constituent databases, federated database systems typically apply wrappers to the subqueries to translate them into the appropriate query languages.

3.2.2 Conceptual foundation of integration of FSPMs

Through the survey of these software integration technologies, we become aware of correlations between these technologies as follows. The layered IOP architecture frameworks give the overall guidance for software integration while the D.S.R ten IOP approaches provide conceptual solutions for each necessary integration aspect. The EIPs embody most of the ten approaches with formal and refined description in icon-based language, and provide technology-independent design guidance for the integration process or workflow. The middleware technologies provide support for the Technical IOP by enabling communication between distributed applications. The component technologies provide support for the Organizational IOP by enabling interactions between different software applications. The data integration technologies provide support for the syntactic and semantic IOP by solving the heterogeneity of data that exists in different data models.

Taking both the correlations and the high complexity of integration of FSPMs revealed by the requirement analysis into account, we conclude that a single technology cannot provide a complete solution for the integration. It is necessary to have concrete technologies including component and middleware technologies, ETL pipeline, as well as technology-neutral methods including the ten IOP approaches and software architectures to provide the overall design guidance for us to design and implement an integration solution with high quality.

Consequently, we determined a list of technologies as the conceptual foundation of the integration of FSPMs based on previously obtained requirements from the available technologies for general integration of software. It includes Webservices technology, RPC middleware model, and K.C. IOP framework, Message Translator & Canonical Data Model design pattern, and ETL data preparing process. We also try to rationalize the logical relationship between these technologies following the D.S.R. ten IOP approaches to make a detailed study of each technology to facilitate the design of our solution for the integration of FSPMs.

3.2.2.1 Determining the conceptual foundation

As one of the two main types of architecture for software integration, different IOP architecture frameworks have been considered, and the K.C. IOP framework is prominent. It clearly distinguishes the levels of interoperability between objects to be interoperated, i.e. signals, data, information, and processes. Compared to the other frameworks that have layers with complex technical nature, e.g. the technical interoperability layer for EIF, this layered structure makes it intuitive and distinct to identify corresponding candidate technologies for each layer. It provides an "object oriented" framework for the interoperability of FSPMs. With it, the correspondence between levels/layers – integrating objects in the framework and the various aspects obtained from requirement analysis can be easily established.

In detail, two aspects, the syntax – semantics aspect and the non-retroactive – retroactive aspect, directly correspond to layers of the framework. The former represents the FSP data and information, and exactly corresponds to the syntactic and semantic interoperability layers of the framework. The latter represents the linkage of processes among different FSPMs, i.e. cooperation at the domain knowledge or semantic level, which corresponds to the layer of organizational interoperability. The other aspects do not correspond to layers of the framework, but reflect the intertwined situation of the layers. The modeling platform – model aspect represents the "vertically" intertwined situation of syntax and semantics of

information in FSPMs. As previous analyzed, in this project, the FSP data of all modules of a plant need to be exchanged together to ensure the topological relations, which reflect also a biological context of a plant module. Therefore, for FSP data of a module, the syntactic interoperability has to ensure the data type compatibility. For data of primary type, most languages have similar sets of data structures, direct mapping or casting can achieve the compatibility, and straightforward technical candidates are available in most programming languages and FSP modeling platforms for this kind of solutions. For the data of composite type, especially for the graphic primitives defined in graphics libraries as a part of a modeling platform, it is much more complicated. A graphic type itself has both a syntactic aspect and a semantic aspect. In most cases, an object-oriented class is used to define such a type. The syntactic aspect of a graphic type refers to the signature of the class constructors, i.e. the types, order of parameters and returns. The semantic aspect of a graphic type refers to the graphical meaning of the name of a constructor (or class). The dependent – independent aspect, the spatial – temporal aspect, the topology – geometry aspect and the internal – external aspect represent the "horizontally" intertwined situation of syntax and semantics of information in FSPMs. Syntax and semantics of functional and structural information of a plant module are two interdependent sides of the information, thus they have to be interoperated simultaneously with all other dependent information. For example, although the different environmental information might be considered independent from each other in FSPMs, the environmental information might still need to be exchanged because there might exist specific dependencies between functional and structural information of a plant module and environmental information. The preparations of plant properties and simulators and the coordination of execution of FSPMs are caused by the independent development of the FSPMs to be integrated. Hence, they correspond to the layer of organizational interoperability. All the correspondences between requirements and IOP layers proves the role of K.C. IOP framework as one of the conceptual foundations of our technology.

It is clear that the integration of FSPMs is primarily a type of software integration, i.e. under heterogeneous technology environments (e.g. different programming languages, different platforms). Therefore, compared to other component technologies, the Webservices are fully in line with the requirements and thus can be determined as one of the conceptual foundations of our technology. Precisely, a web service makes software applications available over networks using standard technologies so that they can perform cross-language/platform interaction. This makes it highly suitable for building distributed integrated FSPM that must incorporate diverse FSPMs over a network. The standard technologies of Webservices include Web Service Description Language (WSDL) specifying how component interfaces should be defined, XML messages formatted with SOAP protocol to communicate with other applications, through a network protocol like HTTP [8]. The WSDL is a standard XML format for describing web services, and is the key to the interoperability of Web service agents. A WSDL file defines a web

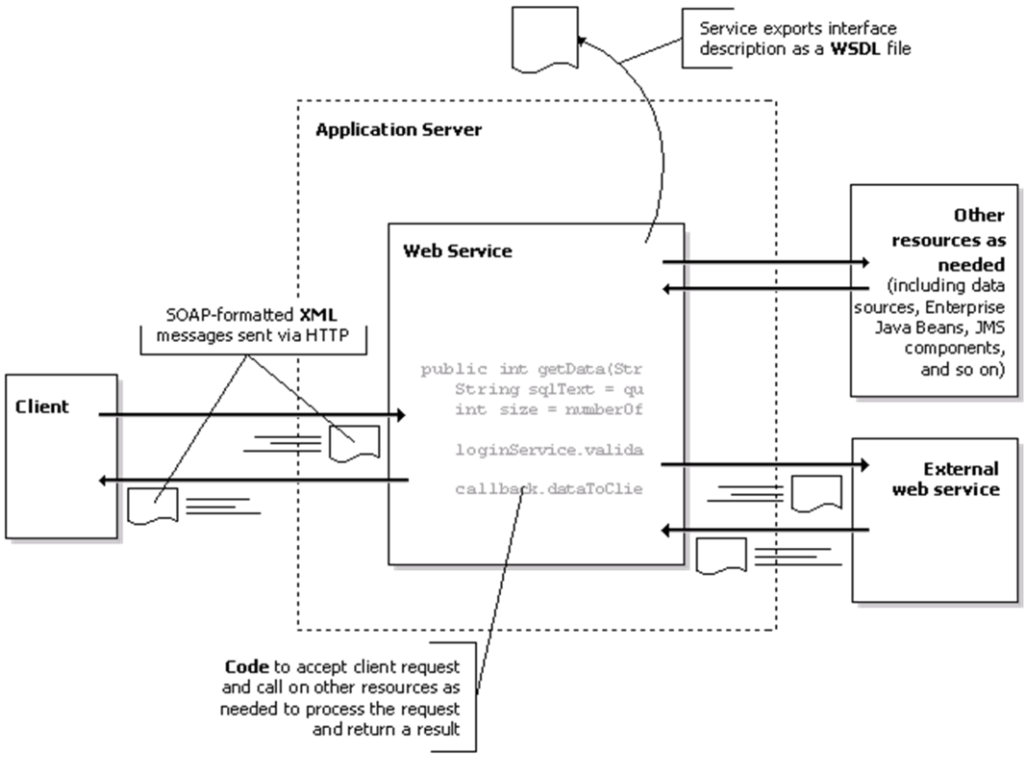


Figure 3.6 Web Service architecture [8]

service as software component by specifying its component interfaces. Other software applications can interact with it by writing components to access the service through the interfaces [137]. The Webservices architecture illustrated by Figure 3.6 [8] clearly shows the relationships between distributed software components and their interaction. In detail, a web service is deployed on an application server responsible for message routing. It interacts with a component of the client software application and resources or external web services through SOAP formatted XML messages over standard protocols such as HTTP.

The requirement analysis implies a client-server relationship between structural and functional FSPMs because a biological structure is assumed as the basis of all its functions. Consequently, the plant structural model always needs to wait for the response of the plant functional model for further execution. This indicates that the RPC middleware technology meets the requirements of the integration of FSPMs and thus can be determined as another conceptual foundation of our technology. In detail, RPC enables request-response or client-server communication between programs over a network without need to understand the lower level protocols. It is based on certain transport protocols, e.g. TCP or UDP, for carrying information/data. The process of a RPC program involves five parts: Client, Client Stub, RPC Runtime, Server Stub, and Server. The Client initiates a RPC by calling the Client Stub with parameters. The Client Stub packs the parameters into a message and passes it to RPC Runtime. The RPC Runtime on the client machine sends the message to the server machine through a communication network. The RPC Runtime on the server machine passes the received message to the Server Stub. The Server stub unpacks the parameters from the message, and then calls the Server procedure. The result replies to the client in the reverse direction. The simplest RPC system is XML-RPC or JSON-RPC using HTTP to transport the message carrying the calls (i.e. method name and parameters) encoded in XML or JSON. Technically, it consists of three parts, including data model (i.e. a set of types for use in messages), request message structures (i.e. an HTTP request with method

name and parameters), response message structures (i.e. an HTTP response with return values or fault information).

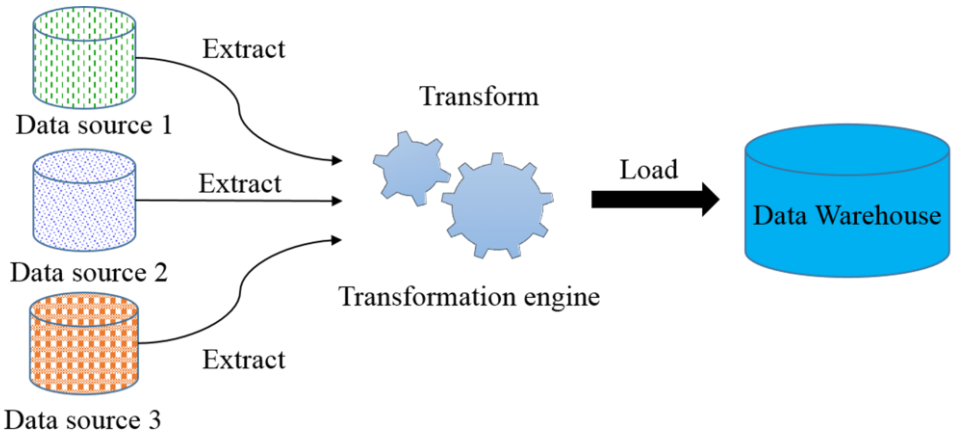


Figure 3.7 Classical ETL Diagram for Data warehouse

Being a specific type of software integration, IOP of information, i.e. syntactic and semantic IOP, are essential for the integration of FSPMs. The requirements indicate that for FSPMs, some specific IOPs, namely the IOPs of plant functional and structural information (including topology and geometry) are needed, and the information heterogeneity needs to be solved physically as the complete information of same plant needs to be transmitted and processed by different FSPMs. As the typical data warehouse technology that provides this physical support, the ETL (c.f. Figure 3.7) [138, 139] data pipeline was determined as one of the conceptual foundations of our technology to support IOP of information. Precisely, data extraction collects data from various sources. Data transformation operates data by converting them into a proper storage structure. Data loading inserts data into the target database (typically a data warehouse). A well designed ETL system extracts data from different sources, ensures data quality and consistency, and conforms data so that various sources can be used together for delivery of data in a uniformed format for the usage of end users. For our project,

the data preparation process of the ETL pipeline embodies the interoperability of the FSP information, is thus useful for the integration of different FSPMs.

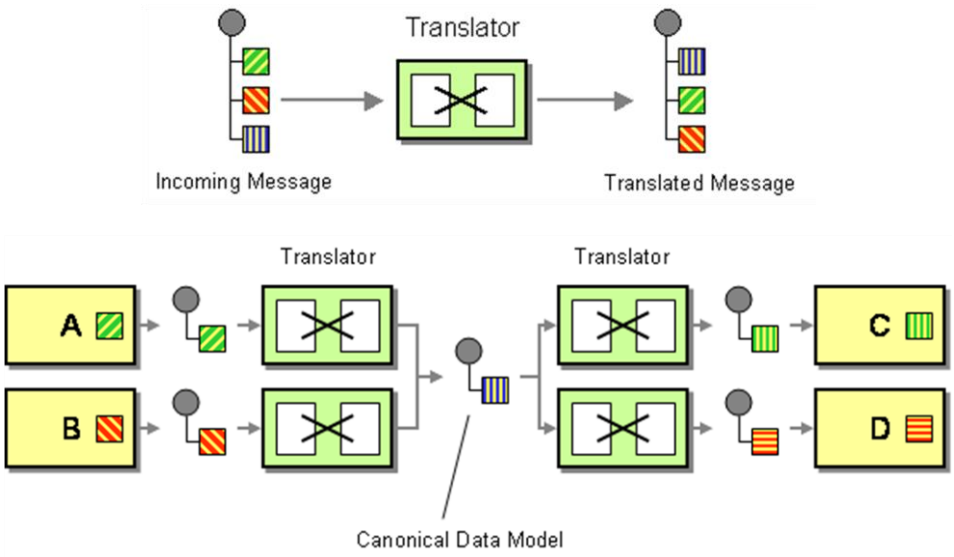


Figure 3.8 Message Translator EIP (upper), Canonical Data Model EIP (lower)[7]

The requirement analysis also indicates the necessity of an intermediate form because of the relation between function and structure, which was embodied by the Canonical Data Model EIP and the approach 7 (*Introduce intermediate form*) of the D.S.R. ten IOP approaches. Thus, this is undoubtedly determined as one of the conceptual foundations of our technology to support the IOP of information. In addition, we have found that the Canonical Data Model EIP has two opposite ETL pipelines made up of four Message Translators (c.f. Figure 3.8), which embody the approach 6 (*provide import/export converters*) of the D.S.R. ten IOP approaches. We thus determined the Message Translator EIP and the approach 6 as supplements to support the IOP of information. Precisely, a message translator [7] is a process translating the messages exchanged between different enterprise applications. The transformation can occur at different levels, including data structure, data types,

data representation, and transport. A canonical data model [7] is a data model that is in the simplest form based on a standard integration solution and independent from any specific application. It provides an additional layer of intermediary between the various data formats/data models of applications. If a new application is added, a development for the transformation between the canonical data model and the individual data formats/data models of the new application is enough, which is independent from the transformations between the canonical model and applications within the existing solution. It is a generic intermediate form providing the integration potential not only for now but also for the future.

Chapter 4

DESIGN OF TECHNOLOGIES FOR THE INTEGRATION

Based on the introduced technologies for software integration, the requirements for the integration of different FSPMs can be rationalized to following aspects. (1) FSP data transfer by HTTP message. (2) FSP graph exchange by an intermediate FSP data model of integrative protocol on top of HTTP. (3) Automatic linkage of FSPM processes/simulators through 'provides/requires' interface of components. (4) FSP graph conversion between intermediate FSP data model and FSP data model of target FSPM by Canonical Data Model EIP with embedded Message Translator EIP that conform to the ETL pipeline and embody the D.S.R. approaches 6 and 7. (5) The preparations for plant properties and simulators of different FSPMs. (6) The coordination for the execution of different FSPMs. It is clear that none of the technologies can cover all the aspects. Consequently, the combination of these technologies is necessary to support the integration of different FSPMs.

In this chapter, we introduce the specific technologies designed for the integration of different FSPMs based on the determined conceptual foundations. This includes a component model based on WebServices technology, a network protocol based on JSON-RPC as middleware technology, and an architecture

framework based on the K.C. IOP framework and Canonical Data Model EIP with embedded Message Translator EIP that conform to the ETL pipeline and embody the D.S.R. approaches 6 and 7 (c.f. Figure 4.1). .

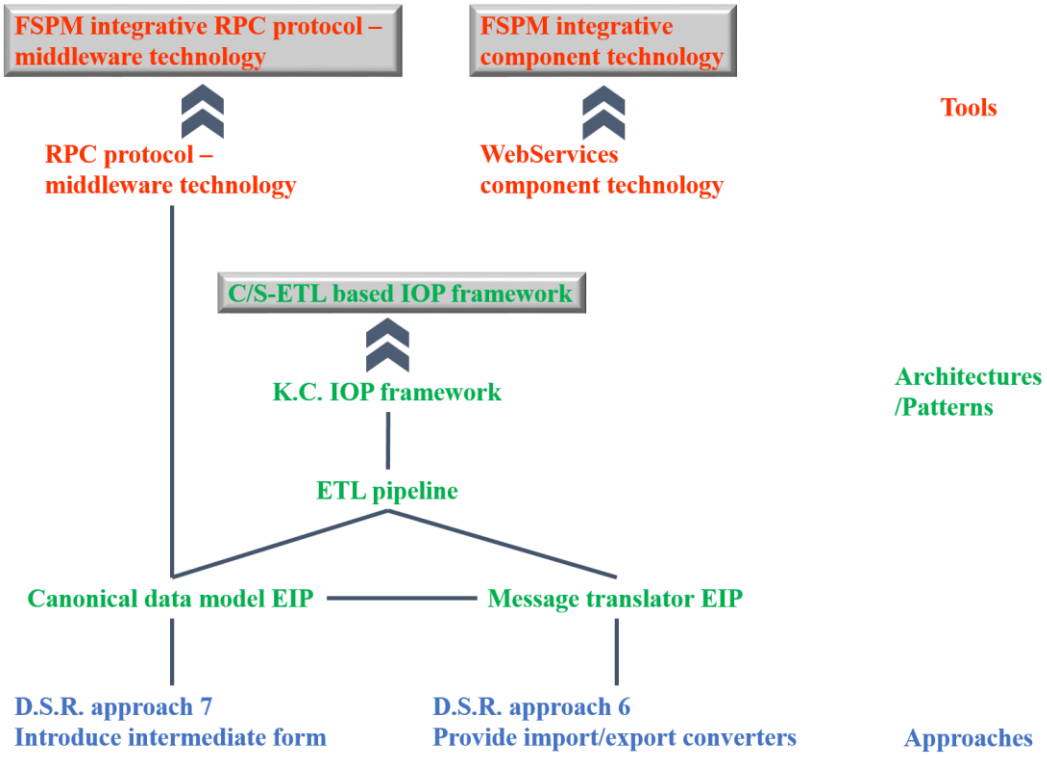


Figure 4.1 Relationships between involved technologies for FSPM integration.

During the technology survey for the integration of different FSPMs, we have noticed some confusing situation/ambiguity between the component technology, middleware technology and software architecture. The situation is that network protocols are often referred to as middleware. For example ORB provided CORBA to allow program calls to be made among distributed software components over network being constantly referred to as middleware, which is actually a feature based on RPC protocol for communication over network. The fact is that the essential functionality of a middleware is to integrate components of application

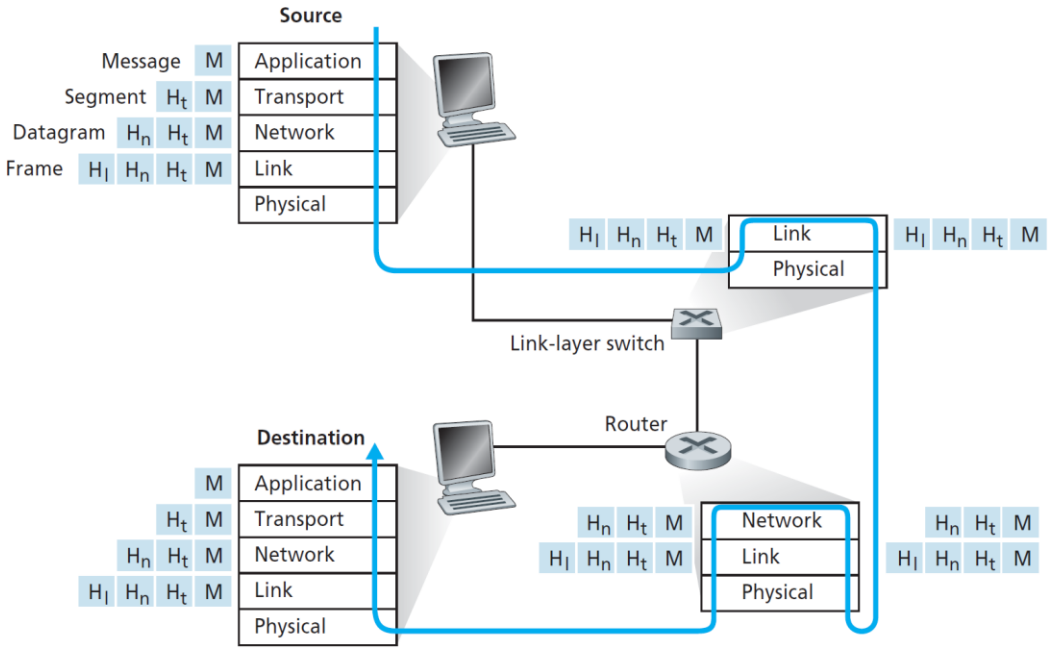


Figure 4.2 TCP/IP protocol stack and data encapsulation [4]

software located on the network. The key point is that the middleware provides services beyond those available from the operation system. According to the mainstream network model, namely the TCP/IP model, different network protocols are actually layered hierarchy. A layer serves the layer above it and is served by the layer below it. Figure 4.2 clearly shows the relationships between layers of different network entities in a TCP/IP network, which are mainly embodied by recursive data encapsulation. For example, the segments of the transport layer encapsulate the message M of the application layer and a header Ht of the transport layer. Consequently, a protocol is a middleware with regard to the layers lower than its reference layer. If a new protocol is designed on top of a protocol that is available from a reference layer, the new protocol is then a middleware with regard to the current reference layer. However, if the protocol is later standardized and becomes available from the updated reference layer, then it is a protocol with regard to the updated reference layer. IETF defines middleware as “those services found above the transport (i.e. over TCP/IP) layer set of services but below the application

environment (i.e., below application-level APIs)" [140] because the support of protocols at transport (and lower) layer is commonly available in operation systems. Another situation is that specific IOP tools are often referred to as implementations of all the three technologies. For example CORBA is referred to not only as component technology and software architecture, but also as middleware technology [141]. The fact is that the three technologies are overlapping in terms of software integration. The component technology supports both software composition and integration, its focus is the independence and reusability of software. The middleware technology is most commonly used in distributed environments to support software integration. Its focus is making complexity caused by distributed environments (such as communication and interoperability) transparent. Therefore, as a concrete component technology, i.e. a component model, the architecture described by CORBA should include the architecture of components, the standard method exposing 'requires' and 'provides' interfaces and allowing the operational interactions between different components, as well as a protocol to provide the communication service to the interaction over a distributed environment. With regard to the conceptual foundation of the integration of FSPMs, the WebServices have the WSDL as standard method to expose 'requires' and 'provides' interfaces. Its SOAP is a protocol, and is a middleware technology as well with regard to operation systems.

Based on the survey, we have also noticed that component modes are highly diverse and can include different combinations of model elements. Table 3.1 clearly illustrates different model elements provided by JavaBean, COM and CORBA. A more complete comparison of the component models can be found in [142]. Some studies have attempted to standardize the component model technology. This includes the standardization of what a component model should describe by summarizing it in a comprehensive list of model elements. For example, model elements have been categorized into six areas, including composition, provided interfaces, dependencies, instantiation, interactions, and assembly [143]. Ian

Sommerville have summarized the basic elements of an ideal component model with even more elements (c.f. Figure 4.3 [1]). Their studies include also the standardization of the way to describe a component model. They suggest to describe a component by three views, including a component diagram as static view that describes relationships between components, an activity diagram as dynamic view that describes interactions between components, and a component description with an appropriate level of details that is related to one of three (conceptual, specification, or physical) levels of model elaboration. [144]. Based on all these studies, we conclude that our technology for the FSPMs integration should be a component model with standardized method to expose interface and component architecture that abstract the needed components and their relationships. A protocol for communication as middleware technology should be established first to support the component model.

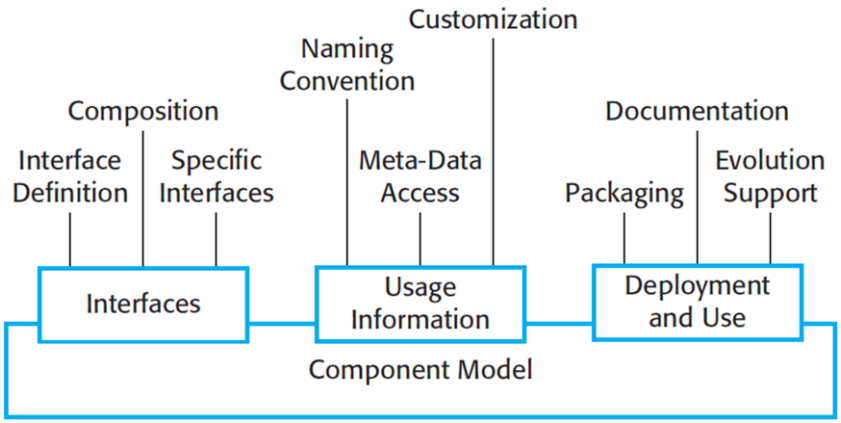


Figure 4.3 Basic elements of an ideal component model [1]

4.1 Design of a middleware technology

It is clear that the difference between software composition and integration is that communication protocols are involved as middleware, which are therefore the basis of software integration. So, a specific protocol suiting the requirements of

FSPMs integration needs to be designed at first. The protocol, as we introduced in the last chapter, is conceptually based on the JSON-RPC protocol and conforms to the Canonical Data Model EIP.

To fulfill the primary needs of information interoperability, we have designed a canonical data model. This is also the key that differentiates our protocol from the existing JSON-RPC protocol. The design includes data exchange models at both logical and physical level. The logical data exchange model defines the logical relations between FSP data elements with simplest abstraction, has high adaptability for different cases of FSPM integration, can be detailed to the physical level to suit specific cases. The physical data model implements the logical structure using s specific serial format to suit the specific needs of our project.

4.1.1 Design of a logical data exchange model

We now introduce the design of a logical data exchange model by adapting our published article [17]. As a canonical data model at the logical level, the logical data exchange model is about to design a logical structure that is simple and abstract enough to enable the data in different FSP data models to be exchanged. Therefore, it should be a generic structure for FSP data organization.

Being a data model abstracting structure and function of plants, the logical data exchange model itself can also be created by data modeling based on the analysis and abstraction of plant structures in the real world. Similar to the other FSP data model, the focus should be on the structure, as “structure is the basis of function; function is the performance of structure”. In essence, there are two levels of requirements for designing a logical data model for abstracting plant structure. The first is the syntactic level, which is the basic level for every kind of data models and which refers to how the elements of data may be organized and accessed. As a plant structure model, some important characteristics of plants need to be taken into account: (1) plant components normally emerge and grow based on existing

components; (2) nutrients reach a component after going through a path consisting of preceding components that are physically connected; (3) also the amount of components and interconnections changes constantly during the whole life cycle of the plant. Because of these characteristics, the elements of plant architectural data are highly connected and codependent with a high rate of change. This demands a data model with high efficiency of update such as insertion and deletion of elements of data.

Apart from the requirements at syntactic level, expressive relationships between elements of data representing dependencies between plant components are biologically meaningful. To automatize various types of biological reasoning, the meaning of the dependencies should also be captured. This leads to the requirement on the semantic level, and demands a data model capable of capturing the semantics of the dependencies/relationships between elements of plant architectural data.

In addition, FSPMs distinguish between function and structure of plants and regard the structure as the basis of the function. Hence, the way organizing elements of architectural (topological or geometric) data of the expected data model needs to be syntactically and semantically different from functional elements. In other words, the architectural data elements are required and the functional data elements are optional. The functional data elements are attached to the architectural data elements. The semantic relationships representing adjacency (i.e. biological dependency) between plant components exist only between architectural data elements.

Structured data models, such as the relational model, do not meet these requirements [145]. The reason is that for elements of plant architectural data, these models are capable of a high efficiency when responding to queries, but have difficulties to capture the semantics of the dependencies, and suffer from a low efficiency update. On the other hand, some semi-structured data models do not distinguish between different elements of data. There is no concept of some

elements of data having more precedence, or importance, over other elements, e.g. properties of a resource are also resources in the Resource Description Framework (RDF) [146], and thus these kinds of data model do not meet the requirement.

The development of plant data models demonstrates an evolution from specific architectural models for specific plant structural modeling, via generic architectural models for structural modeling, to generic FSP data models for FSP modeling. The MTG and RGG graph are two typical data models that are currently widely used and accepted as standards for FSP data modeling, and they are the target data models of our project as well. Hence, the detailed comparative analysis between MTG and RGG graph is helpful to get a logical data exchange model enabling the exchange of FSP data between MTG and RGG graph.

From the facts shown in the detailed comparison of the MTG and RGG graph on both design and implementation introduced in the second chapter, some common elements were discovered and abstracted. At design level, the two data models in their current version are both multi-scaled, with the support of three types of adjacency to abstract the neighboring relationships between modules of real plants. For the MTG, the within- and inter- scale topology are both rooted trees, while the overall topology is a rooted graph. For the RGG graph (i.e. three-part-graph), particularly the instanced graph, the within-, inter- scale, and overall topology are all rooted graphs. Therefore, the topology of the RGG graph is the more general and was considered as the topology of the logical data exchange model. At the implementation level, both MTG and RGG graph are a combination of property graph and scene graph, but with opposite primary/secondary relationship and other specific settings, e.g., geometric data elements can only be represented as properties of graph nodes in the MTG, transformations without other (functional) properties. Thus the logical data exchange model considers the same combination but with no primary/secondary relationship and no specific settings. Topologically, a property graph is a scene graph with properties attached to nodes and edges. For the logical

data exchange model that considers only the topology, the abstraction from the implementation level is thus the “general” or “original” property graph.

The property graph is a type of semi-structured data model distinguishing nodes and their properties. In these logical models or graphs, nodes and edges are used for representing the elements of architectural data and relationships respectively. This makes insertion and deletion of nodes very easy and fast, and ensures a high efficiency update. Moreover, different types of relationships are defined to explicitly describe the meanings of the relationships, so the data model becomes a semantic network and automatic reasoning can be carried out through relationship paths for computing biological variables. Besides, functional data elements are optional and attached as properties of a node of the graph. This guarantees that the architectural data element takes precedence over the functional elements. The property graph meets all the requirements and suits well for the specific focus of corresponding methods abstracting plant architecture, except the capability of multiscale modeling. Consequently, the logical data exchange model should be the combination of the multi-scaled rooted graph and the property graph, with three types of adjacency to abstract the neighboring relationships between modules of real plants. There should also be an unambiguity property for nodes, i.e. id as the unique identifier of a node, as well as for edges, i.e. id as the unique identifier of an edge, source id as the id of the node where the edge starts, and target id as the id of the node where the edge ends. Unlike the MTG and RGG graph, which have their own specific modeling focus, the designed logical data model does not have a specific focus, so that it is able to function as a data exchange model adapting all the logical variants of rooted multi-scaled property graphs.

On one hand, we derived a logical property graph model by specializing the conceptual property graph with some constraints that exist in both MTG and RGG-based graph, or by generalizing MTG and RGG-based graph. Figure 4.4 illustrates two basic types of components, i.e. Nodes and Edges that are directed and labeled

with a “Type” denoting the type of relationship between their source and target nodes. The arrowheads indicate the direction of edges. Both Nodes and Edges have “Ids” in Arabic or Roman numbers and can be associated with properties, which are “key: value” pairs in italics. The key refers to the property id and the value is the property of a particular node or edge. In addition, we added some constraints for the properties, as illustrated in Figure 4.4 [17], two properties “Name” and “Type” which are associated to each node, and the “Type” property attached to each edge that allows a value set including the three standard options (succession, branch, and decomposition). With the additional semantic features, we enhance the ability of our data exchange model to adapt to heterogeneous plant architectural data.

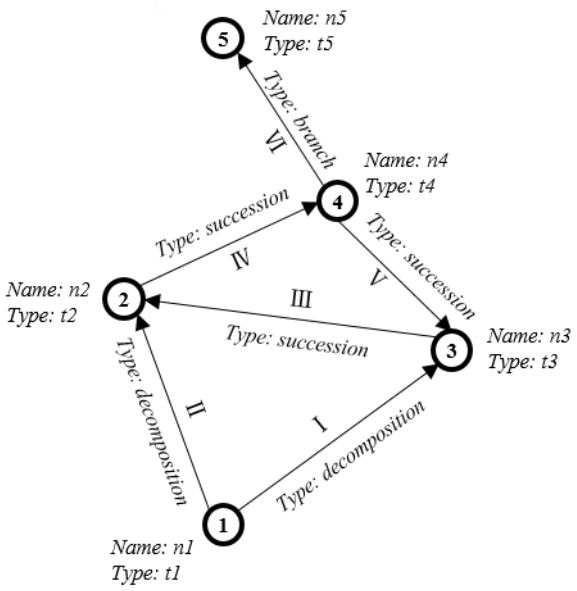


Figure 4.4 Logical property graph model [17]

On the other hand, we propose a logical data model of a conceptual data model “Rooted Graph” [147] [148] with specific constraints. The reason is that we have observed that many applications have a distinguished node serving as entrance node

to their graphs. Figure 4.5 [17] shows our logical rooted graph that is a directed graph in which one single node has been distinguished as the root node (illustrated by a dashed circle). All the other nodes are connected directly or indirectly with the root node. This root node is a special node that does not correspond to any plant architectural component in the real world, but represents the whole plant (or the coarsest scale) for multiscale data.

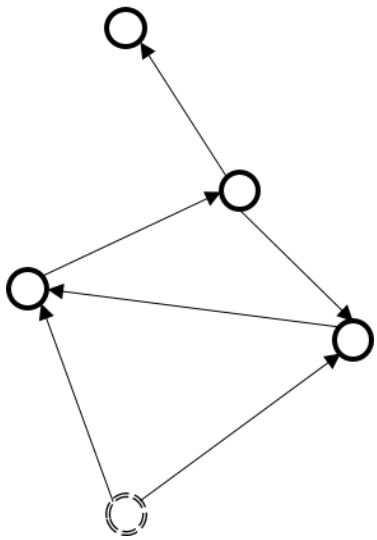


Figure 4.5 Logical rooted graph model [17]

By combining the logical property graph and the logical rooted graph, setting the id of the root node as “root_id” with the fixed value “0” and prohibiting having properties of the root node, we get our targeted logical data model EG (Exchange Graph), as shown in Figure 4.6 [17].

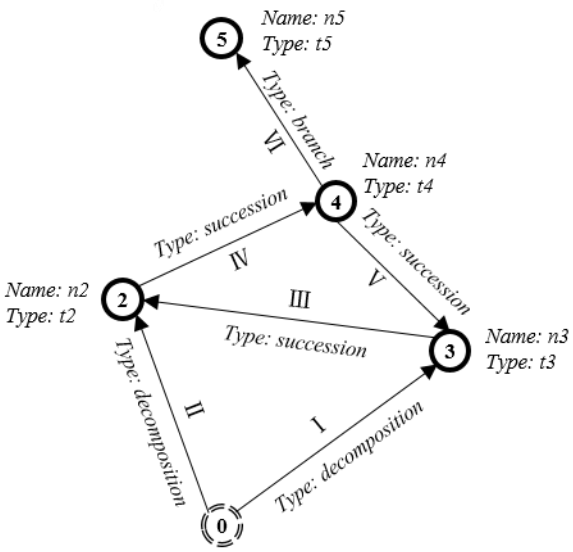


Figure 4.6 Data exchange graph model (EG) [17]

4.1.2 Design of a FSP data exchange model

Based on the EG model, different physical data models can be implemented and enable plant architectural data exchange between heterogeneous FSPMs. We have implemented one in XML because it is an effective tool for standardizing the format of data exchange among various applications, has mature mechanisms for complex data modeling, such as XML schema and DTD for validation.

In detail, our physical data exchange model, XEG (XML based Exchange Graph) has been designed by detailing the logical data model. The full definition of the XEG data model includes its structure, integrity constraints, and applicable operations which are regarded as the import and export modules of the interfacing packages on GroIMP and OpenAlea and are introduced in the next chapter. Imitating the XML-RPC data model that defines a set of general types for use in general messages, we have defined a set of specific types for use in XEG messages.

Using XML elements with different tag names, four XEG elements have been defined to ensure the rooted graph structure:

- An XML element with tag name “graph” represents an XEG graph. This is the highest level of the XML structure. The other XML elements are all nested within it.

- An XML element with tag name “node” or “edge” represents a node or edge of an XEG. A special XML element with tag name “root” represents the root node of the XEG.

Using XML elements or attributes, XEG properties have been defined to ensure the property graph structure:

- An XML element with tag name “property” represents a property of an XEG node. XML attributes “name” and “value” are assigned to the XML element to hold the name and value of the property.

- Some XML elements with specific tag names hold XEG nodes’ property values with complex form as their content, e.g., XML elements with tag name “rgb” holding the XEG node’s “color” property with RGB form (i.e., a numerical sequence with three values representing the red, green and blue component respectively).

- Some XML attributes represent simple properties of XEG nodes, e.g., “id” represents the identifier of an XEG node or edge. The “type” attribute represents the XEG node or edge type. “src_id” and “dest_id” attributes represent the identifier of the XEG node where an XEG edge starts or ends, respectively.

- An XML element with tag name “type” represents the object oriented type extension, allowing a hierarchy of types analogous to a class hierarchy, with inheritance of properties from supertypes to subtypes.

The following XEG code represents a scene with a single green sphere of radius 0.1, possibly representing some plant organs with a functional property ‘p_extended’.

```
<graph>
  <root root_id="0"/>
  <type name="A">
    <extends name="Sphere"/>
    <property name="p_extended" type="float"/>
  </type>
  <node id="1" name="" type="A">
    <property name="radius" value="0.1"/>
    <property name="color">
      <rgb>0.0 1.0 0.0</rgb>
    </property>
    <property name="p_extened" value="0.1"/>
  </node>
  <edge id="1" src_id="0" dest_id="1" type="successor"/>
</graph>
```

Figure 4.7 An example of XEG code representing a plant with a sphere component

Integrity constraints of the XEG data model mainly include the limited choice of the edge type (three basic types: successor, branch and decomposition), the existence and uniqueness of the graph root, the correspondence between “id” of an XEG node and the “src_id” / “dest_id” of an XEG edge. To ensure the validity of the XEG, an XML schema including all the structure and constraints has been defined as XEG schema.

4.1.3 Design of a FSPM integrative protocol

With the designed FSP data exchange model, we can now design the needed protocol-middleware technology, i.e. the FSPM integrative protocol, by referencing the JSON-RPC [149, 150].

The JSON-RPC refers to the RPC protocol that uses JSON to encode its calls and the HTTP protocol as transport mechanism, which indicates that it is an application layer protocol but on top of HTTP. It mainly consists of three specified parts, including a set of data models for typing of data in the HTTP message body, the request and response structures for constructing of the HTTP request and response messages [151]. The data models are data types sharing from JSON, namely four primitive types (Strings, Numbers, Booleans, and Null) and two structured types (Objects and Arrays).

Figure 4.8 Examples of JSON-RPC POST request and response message

The JSON-RPC request structure defines how the method name and its parameters of a JSON-RPC call are packed as an HTTP request message. The HTTP response message structure defines how the returning values or error information of a JSON-RPC call is packed as an HTTP response message. (c.f. Figure 4.8).

By default, the request structure includes:

- A request line with POST as preferred request method
- Request header fields, including at least:
    - Content-Type: value must be application/json

    - Content-Length: value must comply to HTTP protocol specification
    - Accept: value must be application/json
- An empty line
- A message body with a single JSON object consisting of four members:
    - jsonrpc: A string denoting the version of JSON-RPC protocol.
    - method: A string containing the name of the method to be invoked.
    - params: An optional structured value that holds the parameter values to be used during the invocation of the method.
    - id: A client established identifier with String, Number or Null type.

The response structure includes:

- A status line with one of five JSON-RPC specified status codes, or a HTTP specified status code.
- Response header fields, including at least:
    - Content-Type: value must be application/json
    - Content-Length: value must comply to HTTP protocol specification
- An empty line
- A message body with a single JSON object consisting of four members:

- jsonrpc: A string denoting the version of JSON-RPC protocol.
- result: Value is data generated by the invoked method, required on success.
- error: A structured value that holds the parameter values to be used during the invocation of the method, required on error.
- id: An identifier which must be equal to the id of the Request Object.

We have defined our 'FSPM integrative RPC' based on specific assumptions of the integration of different FSPMs. A FSPM, as introduced previously, is a special type of program that can be executed independently. It thus always has a 'main' method, which takes a FSP data graph and some environment arguments as parameters. The integration of different FSPMs is about to allow one FSPM to call the 'main' method of another FSPM over networks in a distributed manner. More specifically in our project, the FSPM for structural simulation has to be integrated with a FSPM for functional simulation by calling its 'main' method. Environmental data will thus only be used as the parameters of the functional FSPM and will affect the structural simulation indirectly by the computed functional properties in the exchange graph received by the structural FSPM. In general, it is hardly true that identical environment parameters are needed for different FSPMs to be integrated. It is because of their heterogeneity that they have the value of being integrated. It is also because of their heterogeneity that their parameters cannot be exactly the same.

With reference to the data models of JSON-RPC, we designed an application layer protocol on top of HTTP with a smaller set of data models as data types. As the XEG is added as a new type for an intermediate form of FSP graph in our protocol, content-type is specified to media type (formerly MIME type) [152] 'application/x-www-form-urlencoded' accordingly. This defines the format of the message body: the keys and values are encoded in key-value pairs separated by '&',

with a '=' between the key and the value. On one hand, JSON and XML are the two most common formats for data exchange, we choose XML as data model format of the XEG for its mature data modeling mechanisms. On the other hand, JSON has less verbose syntax, and is quicker to read and write, the syntax of URL encoded form is even simpler than JSON (e.g. no nested values). The combination of data model XEG and media type 'application/x-www-form-urlencoded' therefore results in a simple but powerful protocol.

With reference to the request and response structures of JSON-RPC, we designed our integrative protocol with simplified request and response structures. Similar to the standard JSON-RPC, a call of our FSPM integrative RPC is represented by sending a set of key-value pairs to a server. The difference is that the key-value pairs in our protocol are not formatted in a JSON object but in a URL encoded form. Moreover, the member of the URL encoded form of the request and response structure have members 'model', 'main_method', 'result_graph' 'time' 'retroactive' to enable the integration of different FSPMs. The value of the member 'model' should be Null when the code of the server FSPM is not available on client side. When a FSPM integrative RPC call is received, a response with a URL encoded form with the result in XEG will be sent back to the client. No specific status codes have been designed at the level of our protocol, and we think the status code mechanism defined at HTTP level is already enough to ensure the correct exchange between different FSPMs.

In detail (c.f. Figure 4.9), the data models of the designed protocol includes four primitive data types sharing from JSON-RPC (i.e. Strings, Numbers, Booleans, and Null), and one structured type (XEG).

The request structure includes:

- A request line with POST as request method
- Request header fields, including at least:

- Content-Type: value must be application/x-www-form-urlencoded
- Content-Length: value must comply to HTTP protocol specification
- Accept: value must be application/x-www-form-urlencoded

- An empty line
- A message body consisting of four members:
  - model: A string containing the code of the FSPM to be integrated. If the code is not sent from the callee, the value must be Null
  - main_method: A string containing the name of the 'main' method of the FSPM to be integrated.
  - graph: An optional structured value that holds the FSP graph (in XEG) as input of the invocation of the method.
  - time: A value of Number type to represent the number of running steps of the target FSPM. It is the key for time scale alignment between different FSPMs.
  - retroactive: A value of Boolean type to represent the retroactive setting of the integration.
  - id: A client established identifier with String, Number or Null type.

The response structure includes:

- A status line with a HTTP specified status code.
- Response header fields, including at least:
    - Content-Type: value must be application/x-www-form-urlencoded
    - Content-Length: value must comply to HTTP protocol specification
- An empty line
- A message body consisting of two members:
    - result_graph: Value is an FSP graph in XEG generated by the invoked method, required on success.
    - id: An identifier which must be equal to the id of the request.

```
POST /myrpc HTTP/1.1
Host: 127.0.0.1:8080
Content-Type: application/x-www-form-urlencoded
Content-Length: 67
Accept: application/x-www-form-urlencoded

model=NULL&main_method=interceptLight&graph=<gra
ph>…</graph>&id=1


HTTP/1.1 200 OK
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 36
Date: Sat, 15 Feb 2019 18:13:55 GMT

result_graph=<graph>…</graph>&id=1
```

Figure 4.9 Examples of the FSPM integrative protocol request (upper) and response (lower) messages.

4.2 Design of a component model

To design a component model for the integration of FSPMs, we need to design at least the component architecture and the standard method to expose the ‘required’ and ‘provided’ interfaces of different FSPMs through the designed middleware, as they are the two basic elements of a component model. As previously introduced, the requirements of the integration of different FSPMs include five aspects and the component model is about to enable the automatic linkage of FSPM processes/simulators through ‘provided’ or ‘required’ interfaces of software components in the integration systems. By referring to the relevant conceptual foundation, i.e. Webservices technology, we designed a standardized component architecture of the FSPM integration system with language neutral method for exposing ‘required’ and ‘provided’ interfaces, which effectively fulfill the required aspects. Beside of the two basic elements, we also provide a detailed component architecture for the middleware of the integration as a standard to facilitate the design and development in integration projects.

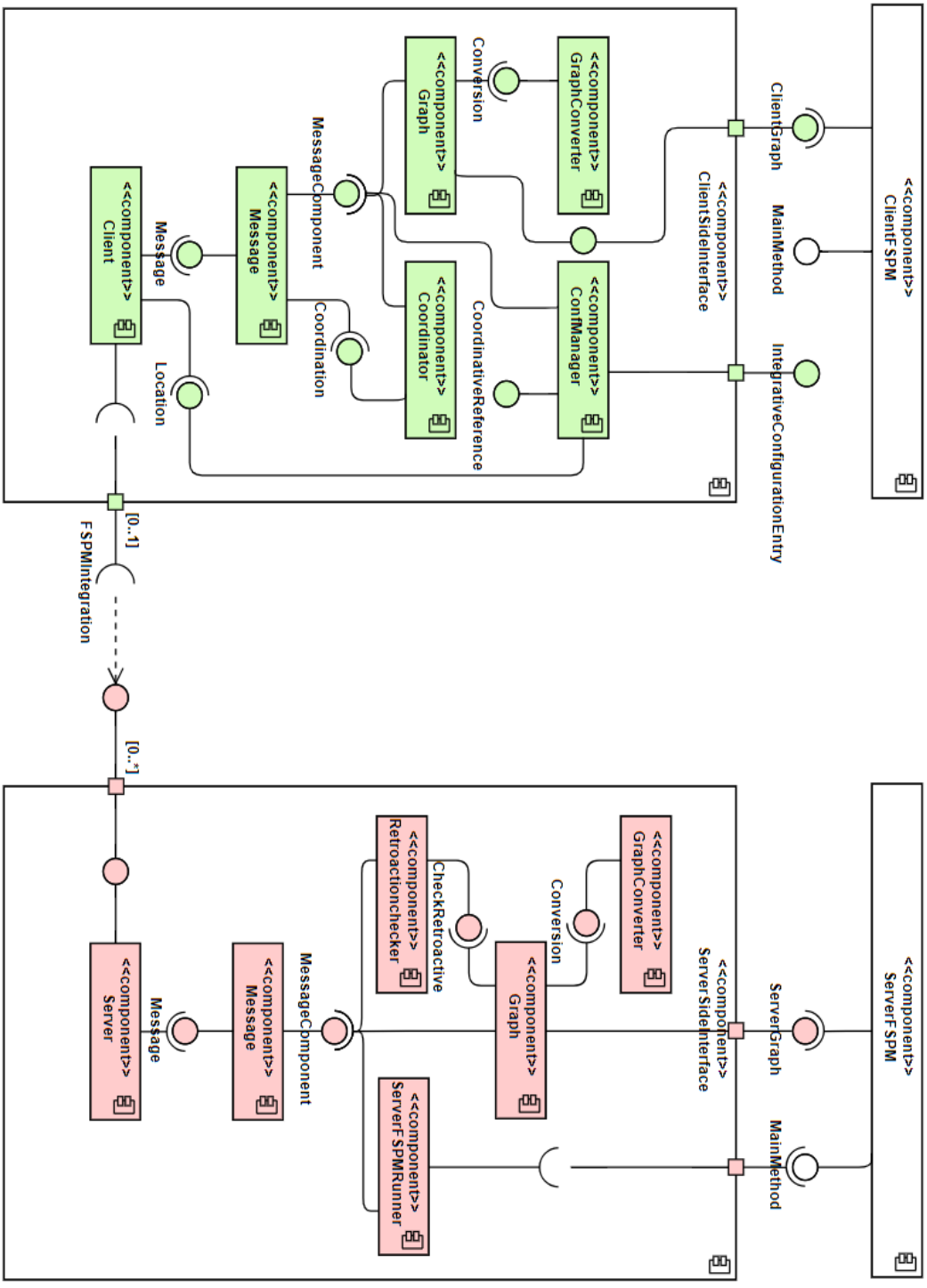


Figure 4.10 The UML component diagram for the integration of different FSPMs

4.2.1 Design of a component architecture

We introduce firstly the component architecture designed based on the analysis of requirements previously introduced and the determined conceptual foundation, i.e. Webservices technology. Its static view is described by the component diagram and component description while the dynamic view is described by the activity diagram.

As the Figure 4.10 shows, the standardized component architecture of the FSPM integration system includes an integrative middleware (we named it FSPM integrative interface) that consists of two components: the *ClientSideInterface* and the *ServerSideInterface*. It resides in the middle of the *ClientFSPM* component and the *ServerFSPM* component to enable their integration by providing necessary interoperability. The component *ClientSideInterface* requires service from the component *ServerSideInterface* for the integration of different FSPMs. The cardinality '[0..1]' at client side means one server may have zero to one client, and '[0..*]' at server side means one client may have zero to many servers. This is a design based on the assumption about the possible integration of different FSPMs scenarios, i.e. one structural model with one or more functional models. The component *ClientSideInterface* includes six components, i.e. *Client*, *Message*, *Graph*, *Coordinator*, *GraphConverter*, and *ConfManager*. The component *ServerSideInterface* includes six components, i.e. *Server*, *Message*, *Graph*, *GraphConverter*, *RetroactiveChecker* and *ServerFSPMRunner*. The components *Message* and *Client*/*Server* are for message packing/unpacking and transmission. The components *Graph* and *GraphConverter* are for the conversion of FSP data and information between different FSP data models. The component *ConfManager* at client side is about to allow the plant scientists to input the configuration setup for the integration based on biological knowledge. This might be a list of model records expressing the order of the model simulation, the name and network address of the model, its 'main' method, the times of its execution, and the names of its

characteristic properties (e.g. ‘interceptedLightAmount’). The component *Coordinator* at client side is about coordinating the simulation/execution of the different FSPMs by taking the configuration list set by plant scientists as reference. By getting relevant member of message body (retroactive) through the provided interface *MessageComponent* of component ‘*Message*’, the component *RetroactiveChecker* is for determining if the client graph is needed to be converted for responding to the client. By getting the relevant members of message body (model, main_method, time) through the provided interface *MessageComponent* of component *Message*, the component *ServerFSPMRunner* is about to run the FSPM on server side through the provided interface *MainMethod* of the server FSPM. One remark is that the data flow in the integrated FSPM is mostly two directional. For example, the ‘message’ interface can be a provided interface of the component *Message* when the message is ready for the component *Client* to send. It can also be a required interface of the component *Message* when the response message is received by the component *Client* and ready for unpacking. We represent these two directional interfaces by one single (one directional) interface for the sake of simplification. Another remark is that the *MessageComponent* interface in the component *ClientSideInterface* includes a ‘two directional’ interface of the component *Graph*, a normal provided interface of the component *ConfManager* and a normal required interface of the component *Coordinator*. They refer to packing or unpacking of an XEG to or from the component *Message*, packing of relevant members of message body (i.e. model, main_method, time, and retroactive) to form the request message, and unpacking of ‘result_graph’ from response message for the use of the component *Coordinator* respectively. The *MessageComponent* interface in the component *ServerSideInterface* includes a ‘two directional’ interface of the component *Graph*, a required interface of the component *ServerFSPMRunner*, and a required interface of the component *RetroactiveChecker*. It refers to packing or unpacking of an XEG to or from the component *Message*, and unpacking of other relevant members of message body

(i.e. model, main_method, time, and retroactive) from the component *Message* respectively.

The static relationships are embodied by the usage dependency between the components. At the level of the FSPM integration system, the interface has two components, *ClientSideInterface* and *ServerSideInterface*. The former requires the FSPM integration service from the latter through the required interface *FSPMIntegration*, namely calls of the designed protocol.

- The component *ClientFSPM*' depends on the service provided through the provided interface *ClientGraph* of the component *ClientSideInterface*,
- The component *ServerFSPM* depends on the service provided through the provided interface *ServerGraph* of the component *ServerSideInterfacde*.
- The component *ClientSideInterface* depends on the service provided through the provided interface *FSPMIntegration* of the component *SeverSideInterface*
- The component *SeverSideInterface* depends on the service provided through the provided interface *MainMethod* of the component *ServerFSPM*
- Plant scientists depend on services provided through the provided interface *IntegrativeConfigurationEntry* of the component *ClientSideInterface* and the provided interface *MainMethod* of the component *ClientFSPM*

At the level of integrative middleware, there are usage dependencies within the component *ClientSideInterface* and the component *ServerSideInterface*. In detail, the usage dependencies within the component *ClientSideInterface* are:

- The component *Graph* depends on services provided through the provided interface *Conversion* of the component *GraphConverter* and the provided

interface *ManageComponent* of the component *Message* (the component *ConfManager* also depends on the service through the latter interface).

- The component *Coordinator* depends on services provided through the provided interface *CoordinativeReference* of the component *ConfManager* and the provided interface *ManageComponent* of the component *Message*.

- The component *Client* depends on services provided through the provided interface *Location* of the component *ConfManager* and the provided interface *Message* of the component *Message*.

- The component *Message* depends on the service provided through the provided interface *Coordination* of the component *Coordinator*.

The usage dependencies within the component *ServerSideInterface* are:

- The component *Graph* depends on the services provided through the provided interface *Conversion* of the component *GraphConverter* and the provided interface *CheckRetroactive* of the component *RetroactionChecker*

- The component *Server* depends on the service provided through the provided interface *Message* of the component *Message*.

- The components *SeverFSPMRunner*, *Graph*, *RetroactionChecker* depend on the service provided through the provided interface *MessageComponent* of the component *Message*

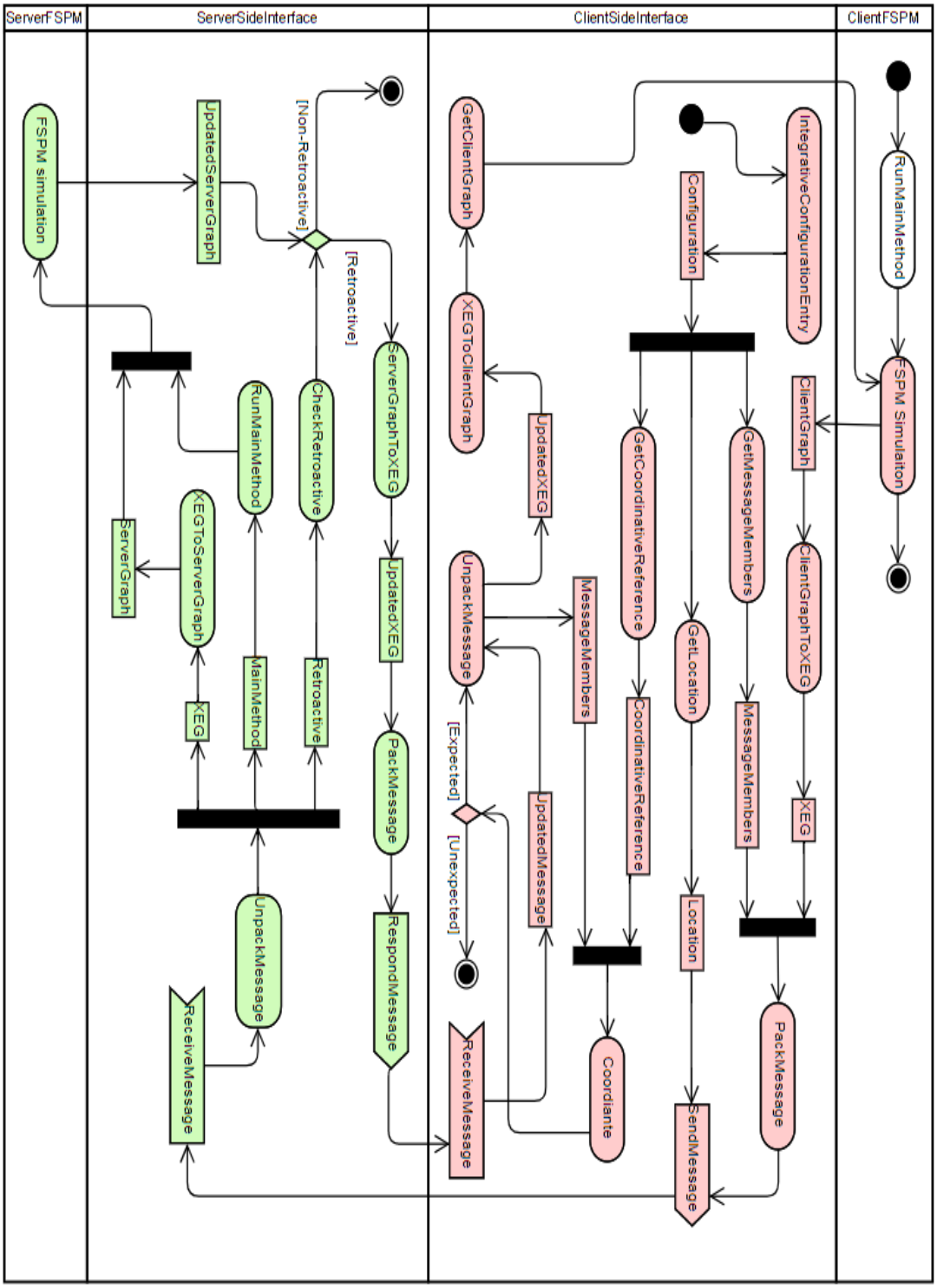


Figure 4.11 The UML activity diagram for the integration of different FSPMs

The dynamic relationships between components are described by an activity diagram (c.f. Figure 4.11). In the diagram, the action flow starts from the *RunMainMethod* action in the *ClientFSPM* partition and the *IntegrativeConfigurationEntry* action in the *ServerSideInterface* partition, the former leads the *FSPMSimulation* action in the *ClientFSPM* partition, which then produces the *ClientGraph* object in the *ServerSideInterface* partition. The actions at the *ClientSideInterface* partition include *IntegrativeConfigurationEntry*, *ClientGraphToXEG*, *GetMessageMembers*, *GetLocation*, *GetCoordinativeReference*, *PackMessage*, *SendMessage*, *ReceiveMessage*, *UnpackMessage*, *Coordinate*, *XEGToClientGraph*, *GetClientGraph*, the involved objects are *Configuration*, *ClientGraph*, *XEG*, *MessageMembers*, *Location*, *CoordinativeReference*, *UpdatedMessage*, *MessageMembers*, *UpdatedXEG*. A decision is made over the result of the action *Coordinate*, which effectively coordinates the simulation of different FSPMs. At the *ServerSideInterface* partition, the actions include *ReceiveMessage*, *UnpackMessage*, *CheckRetroactive*, *RunMainMethod*, *XEGToServerGraph*, *ServerGraphToXEG*, *PackMessage*, *RespondMessage*, the involved objects are *Retroactive*, *MainMethod*, *XEG*, *ServerGraph*, *UpdatedServerGraph*, *UpdatedXEG*. A decision over the result of the action *CheckRetroactive* is made. The action flow ends when the client FSPM simulation is finished, the *UpdateMessage* is determined as unexpected, or the result of the action *CheckRetroactive* is determined as non-retroactive.

4.2.2 Design of a standard to define component interfaces

Beside the component architecture of the integration system, we introduce also the other basic element of our component model, namely the standard to define the interfaces of the components within the integration system based on the relevant conceptual foundation, i.e. the WSDL provided by WebServices technology. By analysis of the WSDL, we found that the designed protocol is exactly a standard way to define the interfaces of the component within the integration system. One

remark is that the interfaces are defined for the interactions between FSPMs on top of the service provided by middleware. It is on the layer of FSPMs that the protocol provides a standard to define the interfaces of components (i.e. FSPMs). Both WSDL and our protocol provide a platform-language independent method to allow the interaction between different FSPMs in a distributed manner. The difference is mainly in who specifies the interfaces when they are applied. It is the service provider who specifies the interfaces in the Webservices technology, while it is the service consumer who specifies the interfaces in our component model.

4.3 Design of a C/S-ETL based architecture

Based on the designed protocol and component model for the integration of different FSPMs, the aspects of the requirements of the integration of different FSPMs can be concretized as : (1) FSP graph transfer by HTTP message, (2) FSP data exchange by XEG as a data model of the designed integrative RPC protocol, (3) automatic linkage of FSPM processes/simulators by the 'model' and 'main_method' provided as members of request structures of the designed integrative RPC protocol, (4) FSP graph conversion between XEG and FSP data model of the target FSPM, (5) the preparation for properties and coordination of simulation/execution of FSPMs by the combination of 'time', 'retroactive' provided as members of request structures of the designed integrative RPC protocol, the component *ConfManager*, and the component *Coordinator* of the designed component architecture, (6) the preparations for simulators of FSPMs.

On one hand, we found that the aspects 1, 2, 3 and 5 are supported by the designed integrative RPC protocol/component model, and correspond to the technical, syntactic, organizational IOP layer of the K.C. IOP framework respectively. On the other hand, we found that the aspects 4 and 6 are not supported by the designed protocol and component model, and they do not have a simple one-to-one correspondence with the layers of the K.C. IOP framework either. This

situation suggests that an IOP architecture framework is necessary as a supplement, and it should be an architecture framework different from the K.C. IOP framework and adapted to the aspects.

With the determined conceptual foundation, namely the K.C. IOP framework and Canonical Data Model EIP with embedded Message Translator EIP that embody the ETL pipeline and the D.S.R approaches 6 and 7, we introduce our designed architecture framework. The essential reason to have an architecture framework is clear: the FSPM integrative component model and the middleware-protocol can only partially fulfill the requirements of the integration of different FSPMs.

To establish our specific architecture framework, we firstly analyzed the weaknesses or defects of the K.C. IOP framework for the integration of different FSPMs in detail. The obvious one is that it does not abstract and reflect all required aspects of the FSPM integration, such as the preparations. Moreover, the preparations are literally needed for organizational reason, but the data/information are involved as objects, which indicate that the IOP layers in the framework overlap each other for the case of FSPM integration. The biggest weakness of the K.C. IOP framework is that, although it perfectly abstracts the IOP related aspects with the four layers, the required aspects of the FSPM integration do not accurately correspond to each layer. For example, the FSP graph conversion between XEG and FSP data model involves both the syntactic and semantic IOP. The inconvenient point here is that the conversion processes are interdependent, it is not appropriate to divide them into different IOP layers. Based on such a situation, we conclude that a framework with a layered structure similar to the K.C. IOP framework is not appropriate. We already know that the Canonical Data Model EIP with embedded Message Translator EIP that embody the ETL pipeline and D.S.R approaches 6 and 7 convert data from one model to another through an intermediate data model. Naturally, we come up with the idea of having our integrative architecture

framework by combining the component model and the middleware-protocol with them. This turns out to be a valid idea: the designed FSPM integrative RPC protocol partially takes the role of the component model and supports the requirement aspects 1-3 the Canonical Data Model EIP with embedded Message Translator EIP that embody the ETL pipeline and the D.S.R approaches 6 and 7 support the aspect 4. This combination of these technologies covers all the aspects except the preparations of simulators of different FSPMs, which cannot be automatized and are one-time processes. We believe it is appropriate to keep the preparations outside, and only present the parts that involve the interactive simulation in the architecture framework. In this way, it can be used directly for the development of the interface/infrastructure without causing misunderstanding. One remark here is that the ETL pipeline, the Canonical Data Model EIP with embedded Message Translator EIP, and the D.S.R approaches 6 and 7 refer to the same essence but from a different point of view, namely the approach, pattern/architecture (c.f. Figure 4.1). In the following introduction, we simply use the ETL pipeline as their reference as it explicitly expresses the processes of data and information.

To have the overall integrative architecture framework, two sub architectures are designed. One is the C/S based sub architecture for the requirement aspects 1-3. Another is the ETL based architecture for the requirement aspect 4. We now introduce the framework by adapting our published article [14]

4.3.1 Design of a C/S based sub architecture

The sub architecture abstracts the aspects 1-3. Within the C/S based sub architecture, a TCP/IP based integrative RPC protocol enables communication and processes cooperation between different FSPMs respectively. The former is automatic while the latter is semi-automatic, which means the cooperation between processes of different FSPMs needs to be ensured by the integrator. For example, a FSPM for simulating plant light interception might run cooperatively with a FSPM

for photosynthesis, not with a FSPM simulating water pressure, because there is a direct biological relation between the first two. Of cause, multiple-integration might enable a chain of FSPMs, in which not everyone has biological relations to all others, but a relation path is needed at least. The C/S based sub architecture only ensures the "how" aspect, not the "what" aspect. So, all of the "what do cooperate" related questions (i.e. domain specific questions) need to be answered and ensured by the integrator before the implementation of the architecture for a specific integration case of different FSPMs.

The main point of the sub architecture is that it consists of one FSPM for simulating structural evolution and one or more FSPMs for simulating function. The reason for this is that a digital plant module representing a real plant organ cannot have more than one pattern of structural evolution. Theoretically, only FSPMs that simulate the structural evolution of the same plant module at different periods or of different plant modules at the same period can be integrated. However, this kind of integration needs a precise plan in advance as accurate time and space need to be aligned while FSPMs were spontaneously created in most cases and were hardly developed with a precise plan for the integration with other specific FSPMs. On the other hand, there is usually no difference between the temporal and spatial resolutions at which functions of a plant are simulated at organ level, thus the time and space (or both) can be coarsely or qualitatively aligned during the integration.

In the sub architecture, the role of client and server was taken by the FSPM simulating the structural evolution and the FSPMs simulating the functions respectively, as "structure is the basis of function". Consequently, multiple servers in our architecture serve one client, contrary to the common C/S architecture that has one server for multiple clients (c.f. Figure 4.12 [14]).

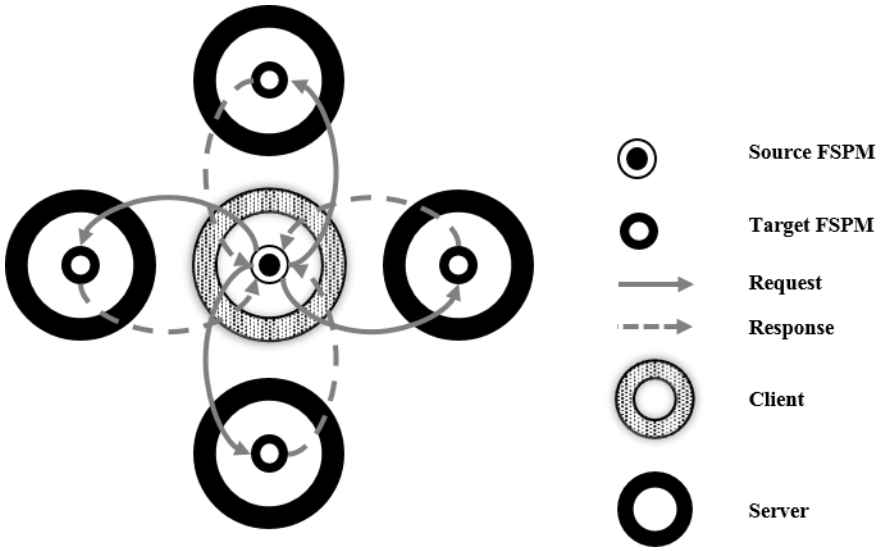


Figure 4.12 C/S based sub architecture [14]

4.3.2 Design of an ETL based sub architecture

Based on the K.C. IOP framework and the Canonical Data Model EIP with embedded Message Translator EIP that embody the ETL pipeline and the D.S.R approaches 6 and 7, we have designed the sub architecture with the ETL pipeline as its focus. The objective of the sub architecture is to enable the FSP information IOP. The sub architecture includes two layers of the overall architecture, namely platform layer and model layer.

The sub architecture includes three data models, the FSP data models of the source and target FSPMs, and the Physical EG, i.e. the canonical data model XEG, as intermediate form. Four message translators consisting of different ETL processes are the main building blocks of the sub architecture, in which the extract and load processes next to the FSPMs are practically import/export converters (c.f. Figure 4.13 [14]).

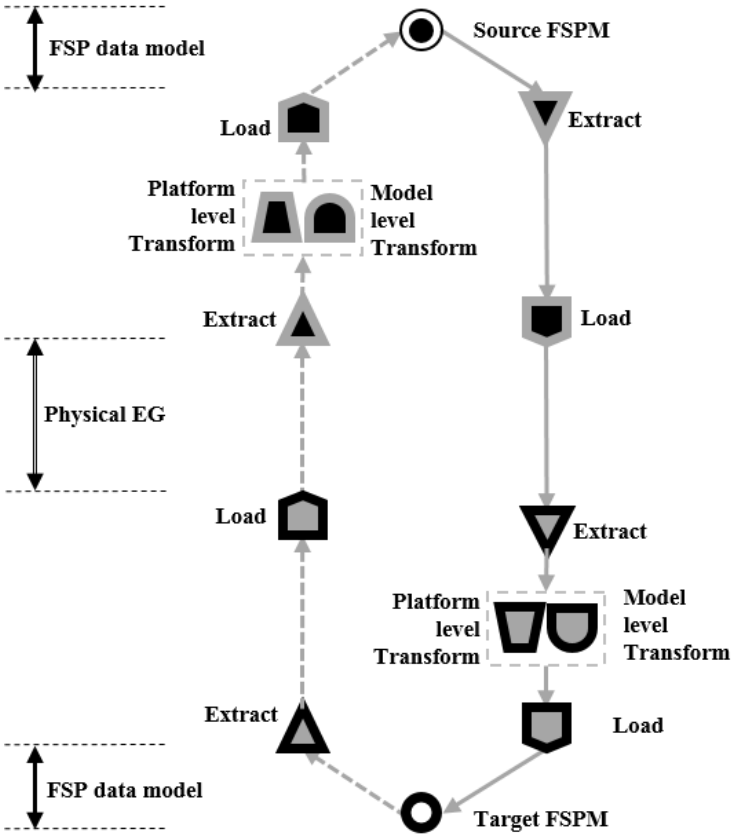


Figure 4.13 ETL based sub architecture [14]

To enable the IOP of FSP information at platform level, ETL processes have to be defined according to the data models and graphics libraries of source and target FSPMs.

For the extract and load processes at platform level, the intra-scale structure is mostly concerned because it is defined as the basic part of the FSP data model at the platform level. Unlike in data warehousing where only data of primitive type are extracted and loaded, the interoperability of information for the integration of FSPMs usually requires more extract and loading data of composite type, e.g., graphics types.

For the transforming process at platform level, several sub processes are necessary to meet the requirements of the integration. (1) Syntactic and semantic transformation of topology of data elements, e.g., generation of an edge in the target graph between corresponding source and destination nodes, and assignment of an edge type according to the edge type in the source graph, e.g., assignment of the "refinement" type available in the target FSP data model to generate an edge of the "decomposition" type in the source graph. Essentially, the topology here concerns the structure with equivalent systems of scales (spatial resolutions). (2) Semantic transformation of the geometry of data elements. This may include a sub process that transforms geometric transformations between local and global. This sub-process is then essentially converting geometric relationships between nodes and is based on graph traversal. To avoid double running of graph traversal, this process is better run with the corresponding extract process. Another sub process performs syntactic and semantic transformations of shape instances, e.g., transforms a signature with argument values of "Parallelogram" type to a signature with argument values of "TriangleSet" type. To allow this sub process, a "dictionary" to "translate" types from the graphics library used in the source FSPM to types in the graphics libraries used in the target FSPMs will be necessary.

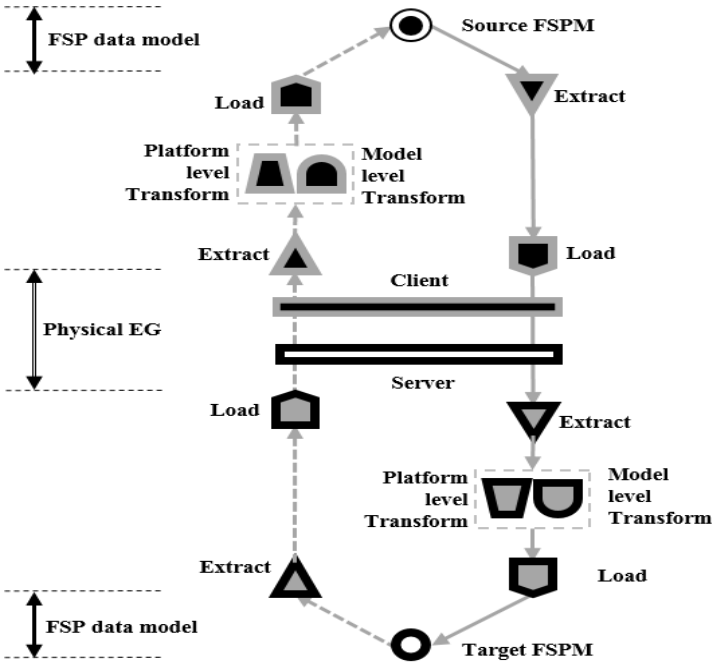
To enable the IOP of FSP information at model level, ETL processes have to be defined also for functional information.

The functional information is usually specified in a FSPM using primitive types (e.g. float or integer). Thus, extract and load processes are not really needed. The transformation process includes two sub processes. (1) Syntactic and semantic translation of coded values or derivation of new calculated values for functional or environmental data fields (e.g., float_ Fahrenheit = (float) (double_ Celsius * 1.8 + 32)). (2) Syntactic and semantic transformation of different systems of scales, for multiscale structures (e.g., decomposition of a scale with metamers as nodes to a new scale with elementary geometric objects as nodes) defined in different FSPMs.

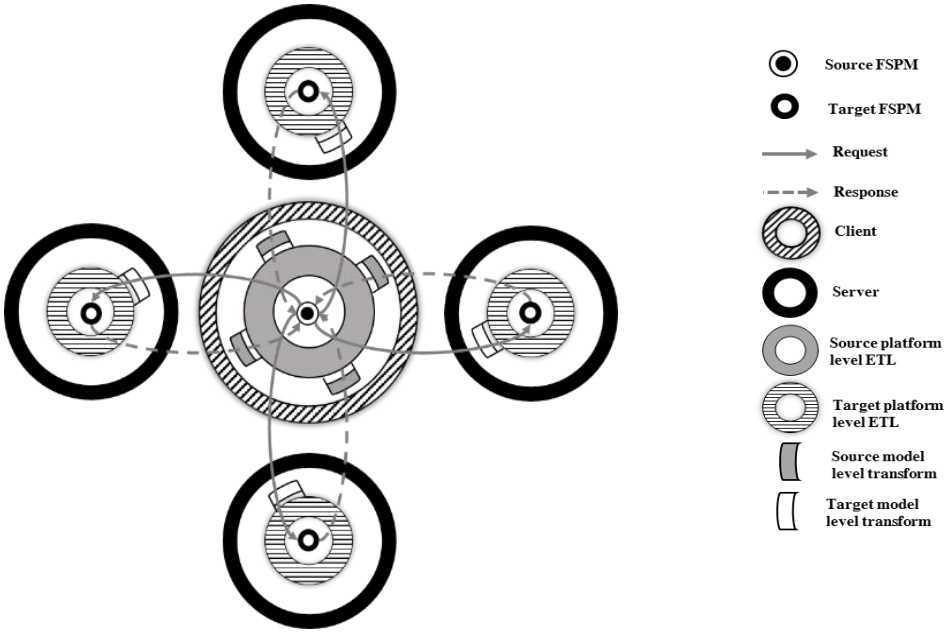
In most cases, simulators are applied to the finest scale of the multiscale graph. Thus the received structure needs to be transformed into a new structure with the finest scale on which the target simulator can be applied. (3) Manual modification of the semantic definitions in target FSPMs if it is necessary. This includes the previously introduced preparations for both FSP information and simulators.

4.3.3 The overall integrative architecture

The overall architecture framework for the integration of different FSPMs (c.f. Figure 4.14 [14]) results from combining the two sub architectures. To be clearer, we present it with both pairwise view and one-to-many view. With the architecture, we can see that the ETL processes are the core of the integration. Particularly the processes at platform level are should be automatized, thus the integration interface should be the infrastructure that implements these processes.



A. Pairwise view of the architecture.



B. One-to-many view of the architecture.

Figure 4.14 Overall architecture framework for the integration of different FSPMs [14]

Chapter 5

AN INTERFACE FOR THE INTEGRATION OF THE TARGET FSPMS

In this chapter, we introduce an interface for the integration of the two FSPMs for the FSPM Apple project, namely MAppleT and the GroIMP based water and sugar transport model, by applying the technologies designed for the integration of different FSPMs, i.e. the FSPM integrative component model, RPC protocol-middleware, and the C/S-ETL based architecture. The design of the interface is focused on the algorithms for IOP of FSP data and information. The implementation of the interface is focused on the software components that conform to the designed technologies for the integration of different FSPMs. The former is about the conception of the ETL processes based on the Breadth-First Search (BFS) algorithm to bridge the differences at both platform (c.f. 2.5.2) and model level (c.f. 3.1.2). The latter is mainly about the realization of the integrative RPC protocol and the component architecture of the integrative middleware/interface based on the two platforms to allow FSP data transfer, FSP graph exchange, and process linkage and coordination between the two FSPMs.

5.1 Design and implementation of the component *ClientSideInterface*

In our designed component architecture of the FSPM integrative interface, the component *ClientSideInterface* consists of six components, which can be divided into a group for communication and interaction between FSPMs and a group for ETL processing. The former group includes the component *Client* and the component *Message* as implementation of the FSPM integrative protocol for constructing & sending the integrative simulation request to the server and receiving & deconstructing the simulated response from the server. As a part of the component *ClientSideInterface*, the group should also include the component *ConfManager* and the component *Coordinator* for the preparation for plant properties and the coordination of execution of FSPMs. We have just implemented GUI modules for the entry of the name and network address of the model, its 'main' method, the number of its executions, and the names of its characteristic properties. This enables the message to be transferred through HTTP, the members of message body to be packed, the preparations for plant properties, i.e. adding properties (i.e. data fields for water and sugar flux and the modification done by MAppleT) to allow that the results of functional simulation can be stored. In our project, only two target FSPMs are involved, the coordination for execution of more than two FSPMs is actually not necessary, and the preparation of simulators of different FSPMs to allow specific functional properties to affect the structural evolvement of apple tree has to be achieved manually. The latter group includes the component *Graph* and the component *GraphConverter* for FSP graph conversion between XEG and MTG. These are the main parts that correspond to the ETL pipeline and are in the focus of the introduction.

5.1.1 The communication group at client side

In the communication group of the component *ClientSideInterface*, the component *Client* is for sending and receiving the HTTP message based on the specification of the HTTP protocol. The component *Message* is for request message construction and response message destruction based on the specification of the FSPM integrative RPC protocol. In our specific case, we have only one server FSPM so the component *ConfManager* for manual entry of configuration information and FSPM related information (location, name of server FSPM and its 'main' method) extraction and the component *Coordinator* for coordination of the interaction of different FSPM processes can be simplified to a manual entry for FSPM related information. The focus of this group is thus the components *Client* and *Message*.

In detail, we have implemented modules that take the combination of IP address and port number (i.e. Socket) to identify the server FSPM on a specific device, the 'model', 'main_method', 'graph', 'time', 'restorative' and 'id' of the FSPM integrative protocol members. All these parameters are taken by GUIs from entry of users except the 'id', which is automatically generated. Moreover, a module that encodes the protocol members into URL form and a module that establishes an HTTP connection with location, sends the form and necessary headers in a POST request and waits for response on this connection have been implemented as well.

5.1.2 The ETL group at client side

In the ETL group of the component *ClientSideInterface*, the components *Graph* and *GraphConverter* are for ETL processes between XEG and MTG. Among the three processes of ETL, the extract process is driven by a graph traversing algorithm and the transforming and load processes are carried out according to the extracted FSP data. The extract process is based on an implemented XEG library in Java and

the focus of extract and load processes is to ensure the correctness of the graph topology. The focus of the transforming processes is to ensure the correctness of the geometry encoded in the graph nodes. The data of each data field of FSP graphs need to be extracted, transformed, and loaded according to the modeling platforms on which the two FSPMs are based, namely OpenAlea and GroIMP. This means the processes might be applicable for the integration of other FSPMs based on the two modeling platforms. Beside the topology and geometry, other kinds of data fields such as colors have been considered as well.

To allow the client side to make the RPC call, we have designed an algorithm to extract FSP data from an MTG (generated by MAppleT) and load it into an XEG, and a set of algorithms to transform nodes of OpenAlea types to nodes of GroIMP types. To allow the client side to process the response of the RPC call from the server side, we have designed an algorithm to extract FSP data from an XEG and load it into an MTG, and a set of algorithms to transform nodes of GroIMP types to nodes of OpenAlea types. Transforming processes between global and local transformations applied to a shape need to be carried out during an iterative graph traversal from parents to children, which confirms the correctness of using the BFS as the algorithm for the extract processes. The implementation of ETL processes at client side has its focus mainly on the transforming processes of nodes of different data types. For the MTG to XEG direction, we have divided the nodes into different categories according to their data types and developed the transforming modules by applying an implantation template designed for nodes of the same category. For the MTG to XEG direction, only nodes of types used in MAppleT are processed and node-transforming modules for these types are implemented following designed algorithms.

5.1.2.1 Algorithms for ETL processes from MTG to XEG

As previously introduced, the FSP data model MTG of the OpenAlea platform lets every node in the MTG correspond to a single or group of plant modules.

Particularly in MAppleT, the topology of the MTG has four different scales, namely the tree, axis, growth unit, and metamer scale. All nodes in the MTG might have biological properties, but only the nodes at metamer scale have graphical properties. In other words, the metamer scale is the primary scale that MAppleT generates first and the other scales are generated based on it. Hence, the algorithm of extract and load needs to have one part for the scales above the metamer scale and one part for the metamer scale. The former part is just to duplicate the topology of the above-metamer scales, while the latter part needs to be designed by referring to the details of MAppleT.

In MAppleT, three 3D shape types (*Cylinder*, *BezierPatch*, *Sphere*) and three transformation types (*Scaled, Oriented, Translated*) of the PlantGL library [110] are used to form the graphical properties (i.e. 3D graphical elements) of a node at metamer scale. As introduced in section 2.5.2, the manner of applying transformations to a shape in PlantGL is taking an object of shape or transformation types as argument to instantiate the object of transformation types. Precisely, there are three patterns of transformation application, which generate three kinds of transformed shapes:

- *Translated (Oriented (Cylinder))*, expresses a transformed cylinder, namely an object of '*Cylinder*' type successively transformed by a rotation and a translation.

- *Translated (Oriented (Scaled (Sphere)))*, expresses a transformed sphere, namely an object of '*Sphere*' type successively transformed by a scaling, a rotation and a translation.

- *Translated (Oriented (Scaled (Scaled (BezierPatch))))*, expresses a transformed object of BezierPatch type, namely an object of '*BezierPatch*' type successively transformed by two scalings, a rotation and a translation.

Based on the three kinds of transformed shapes, MAppleT defines four different kinds of graphical elements as apple tree modules, which are graphic properties of nodes at metamer scale of the MTG. They are not managed by the MTG, but by an object of PlantGL type *Scene* in a list structure and are globally transformed from the origin of a global coordinate system:

- The 'internode' graphic element that is a transformed cylinder. A node with this kind of graphic element as property is referred to as 'internode' metamer node.

- The 'leaf' metamer graphic element that consists of two transformed objects of *Cylinder* type as an internode and a petiole, one transformed object of *BezierPatch* type as a blade. A node with this kind of graphic element as property is referred to as 'leaf' metamer node.

- The 'flower' metamer graphic element that consists of two transformed objects of *Cylinder* type as an internode and a petiole, ten transformed objects of *Cylinder* type as middle of flower, five transformed objects of *BezierPatch* type as petals. A node with this kind of graphic element as property is referred to as 'flower' metamer node.

- The 'fruit' metamer graphic element that consists of two transformed objects of *Cylinder* type as an internode and a petiole, one transformed object of *BezierPatch* type as blade, one transformed object of *Sphere* type as fruit. A node with this kind of graphic element is referred to as 'fruit' metamer node.

In MAppleT, the metamer scale is the primary and finest scale. Moreover, the MTG locates the graphic properties outside the graph, in a data structure called Scene as a type available in PlantGL. It is an object different from the object of MTG type, and unlike a typical scene graph, graphic objects are managed in this

structure with no explicit topological relationships in-between. Our XEG has all kinds of properties for each node, including graphic shapes and transformations. It is a rooted graph managing all data fields, between which there should be at least one topological relationship from the graph root. Consequently, it is necessary to have an additional scale to load nodes of graphic types extracted from the object of Scene type. Thus we perform effectively a decomposition of the metamer scale to a new scale (which we denote as submetamer or organ scale) consisting of elementary graphic objects as nodes. For each of the four graphic elements, we designed a specific decomposition scheme:

- 'internode' metamer node />Orientation>Translation>Cylinder
- 'leaf' metamer node />Orientation>Translation>Cylinder[Orientation>Translation>Cylinder>Scale>Scale>Orientation>Translation>BezierSurface]
- 'fruit' metamer node />Orientation>Translation>Cylinder[Orientation>Translation>Cylinder>Scale>Scale>Orientation>Translation>BezierSurface][Scale>Orientation>Translation>Sphere]
- 'flower' metamer node />Orientation>Translation>Cylinder[Orientation>Translation>Cylinder][Orientation>Translation>Cylinder][Orientation>Translation>Cylinder][Orientation>Translation>Cylinder][Orientation>Translation>Cylinder][Orientation>Translation>Cylinder][Orientation>Translation>Cylinder][Orientation>Translation>Cylinder][Scale>Scale>Orientation>Translation>BezierPatch][Scale>Scale>Orientation>Translation>BezierPatch][Scale>Scale>Orientation>Translation>BezierPatch][Scale>Scale>Orientation>Translation>

BezierPatch][Scale>Scale>Orientation>Translation>
BezierPatch][Orientation>Translation>Cylinder>Scale>Scale>Orientati
on>Translation> BezierPatch]

The algorithm to extract FSP data from data fields of the MTG and load them into data fields of the XEG by constructing an additional scale (i.e. sub metamer scale) using the graphic properties of nodes at metamer scale lets the other properties of nodes at metamer scale unchanged. The topology of the above-metamer scales remains unchanged as well. Overall, the extract and load processes from the MTG to the XEG have mainly the purpose to insert all graphic objects managed by the object of *Scene* type into the original topology as a new finest scale (i.e. a fusion of a pair of objects of *Scene* and *MTG* types) (c.f. Figure 5.1). Its essence is topological downscaling.

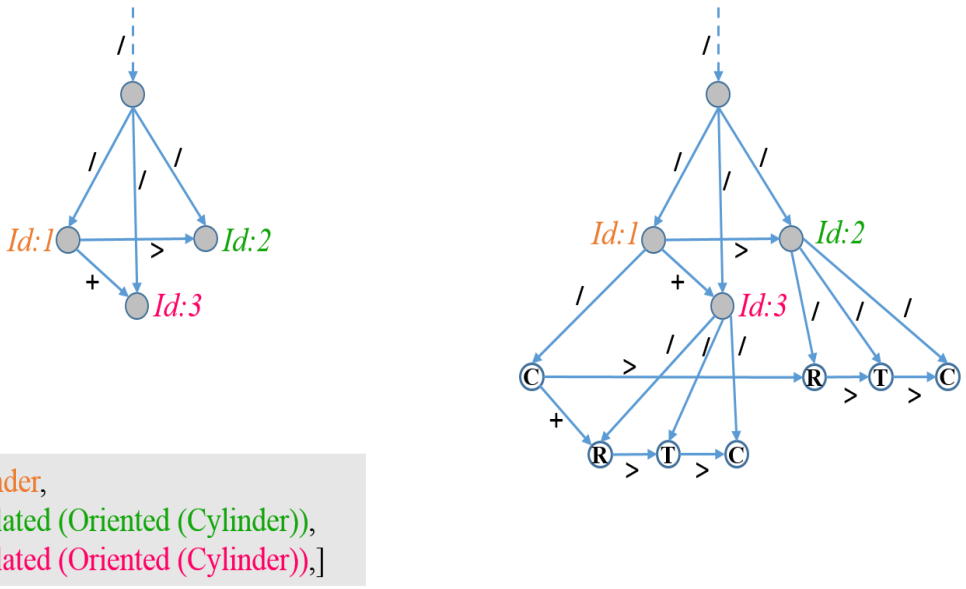


Figure 5.1 Map for fusion of an object of *MTG* type (top left) and a corresponding object of *Scene* type (bottom left) to an XEG (right). The items in the list of the latter object link to the nodes of former object by Ids. R, T, C are rotation, translation, cylinder objects converted from the list items.

From the designed C/S-ETL based architecture, we know that the ETL pipeline from MTG to XEG includes only extract and load processes, which perform essentially a topology conversion. In practice, we have also included transforming processes. The reason it that the nodes at the sub metamer scale are designed to be objects of shape or transformation types, but graphic types of PlantGL and the way the objects are managed are not compatible with the XEG syntax. The graphic types of PlantGL require nested graphic objects, i.e. the nodes at the sub metamer scale must be able to have nodes as properties. This violates the principle of the EG and thus is not compatible with the XEG syntax. Moreover, the objects of shape and transformation types are managed in a set, where there are no topological relationships or edges between the objects, this is also not compatible with the EG's property graph nature. To solve the issue, we placed the transforming processes designed at the server side to the client side. This refers to the transforming processes from nodes of types available on OpenAlea to nodes of types available on GroIMP. As only three graphic types are used in MAppleT, putting the transforming processes on the client side will not cause much substantial impact. Moreover, putting all processes that are applicable for a specific FSPM together makes the implementation modular. We thus believe such a practical adjustment is appropriate.

In this project, there are necessary transforming processes for MTG vertices, shapes, transformations, and colors. We have treated them differently. (1) The part of the MTG that is supposed be loaded to the metamer and above-metamer scales of the XEG are nodes with properties of non-graphic types. They are located in the object of OpenAlea *MTG* type textually as entries of a nested Python dictionary. Hence, they are not typed and it is necessary to assign a data type to them. We have made a string 'MtgVertex' as the type value of these nodes in the XEG. The properties of the nodes are stored as the properties of the XEG nodes without effective changes. (2) The part of the MTG supposed be loaded at the sub metamer scale are graphic object managed by an object of PlantGL *Scene* type. The managed

data include objects of PlantGL types for shape, transformation, color. Actually, the conversion between nodes of two sets of types available in two graphic libraries is about the conversion of their properties according to the signatures of their types. As a graphic type normally has more than one equivalent type signature, technologies that allow conversion between nodes of types with multiple signatures are needed. We introduce them later in the next sections. The focus here is about the algorithms and correspondences between the types used in MAppleT and the types available in the IMP3D library of GroIMP:

- The first step is to compute the individual transformation matrix applied to a shape according to the relevant PlantGL type definition.
    - Get the transformation matrix of a MTG object *od* of *Oriented* type, which represents a rotation:

$$O_m = \begin{bmatrix} P.x & S.x & T.x & 0 \\ P.y & S.y & T.y & 0 \\ P.z & S.z & T.z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$P = od.primary, S = od.secondary, T = P \times S$$

    - Get the transformation matrix of a MTG object *td* of *Translated* type:

$$T_m = \begin{bmatrix} 1 & 0 & 0 & T[0] \\ 0 & 1 & 0 & T[1] \\ 0 & 0 & 1 & T[2] \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$T = td.translation$$

    - Get the transformation matrix of a MTG object *sd* of *Scaled* type:

$$S_m = \begin{bmatrix} S[0] & 0 & 0 & 0 \\ 0 & S[1] & 0 & 0 \\ 0 & 0 & S[2] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$S = sd.scale$$

- The second step is to compute the global transformation matrix *Gm* applied to a shape according to the patterns of transformation application valid for the organ shapes used in MAppleT:
    - The transformation matrix applied to a MTG object of *Cylinder* type is the result $O_m \cdot T_m$
    - The transformation matrix applied to a MTG object of *Sphere* type is the result $S_m \cdot O_m \cdot T_m$
    - The transformation matrix applied to a MTG object of *BezierPatch* type is the result $S_{m1} \cdot S_{m2} \cdot O_m \cdot T_m$
- The third step is to compute the local transformation matrix $p2c_{\text{localm}}$ from location of a parent shape ($parent_{location}$) to location of a child shape ($child_{location}$) using the global transformation matrices ($parent_{global}$) and ($child_{global}$) applied to them:
    - $p2c_{\text{localm}} = parent_{globalm}{}^{-1} \cdot child_{globalm}$

$$= \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$child_{location} = child_{globalm} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

$$parent_{location} = parent_{globalm} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

- The fourth step is to decompose the local transformation matrix to a sequence of objects of transformation types available in the IMP3D library according to the shape to which the transformations are applied.
    - If the transformation matrix is applied to a MTG object of *Cylinder* type, the sequence is *ShadedNull* object>*Translate* object, here the ShadedNull object represents a rotation.
    - If the transformation matrix is applied to a MTG object of *Sphere* type, the sequence is Scale object>*ShadedNull* object>*Translate* object.
    - If the transformation matrix is applied to a MTG object of *BezierPatch* type, the sequence is *Scale* object>*Scale* object>*ShadedNull* object>*Translate* object.
- The fifth step is to form an XEG node for each transformation object of a type in PlantGL using an appropriate type in the IMP3D library:
    - The object of type *Oriented* forms an XEG node with type value 'ShadedNull', and a property with name 'transform' and value $\begin{bmatrix} a & b & c & 0 \\ e & f & g & 0 \\ i & j & k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.
    - The object of type *Translated* forms an XEG node with type value 'Translate', and a property with names 'translateX', 'translateY' and 'translateZ', and their values 0 (c.f. remarks following later).

- The object of type *Scaled* forms an XEG node with type value ‘Scale’, and a property with names ‘scaleX’, ‘scaleY’ and ‘scaleZ’, and their values 1 (c.f. remarks following later).

- The sixth step is to form an XEG node for each shape object of a type in PlantGL using an appropriate type in the IMP3D library:

  - For a shape *cylinder_object* of type *Cylinder*, an XEG node with type value ‘Cylinder’, a property with name ‘radius’ and value *cylinder_object.radius*, a property with name ‘length’ and value *cylinder_object.height*.

  - For a shape *sphere_object* of type *Sphere*, an XEG node with type value ‘Sphere’, a property with name ‘radius’ and value *sphere_object.radius*,

  - For a shape *bezierPatch_object* of type *BezierPatch*, an XEG node with type value ‘BezierSurface’, a property with string ‘data’ as name and the float list converted from *bezierpatch_object.ctrlPointMatrix* as value, a property with string ‘dimension’ as name and the dimension number of the *bezierPatch_object.ctrlPointMatrix* as value, a property with string ‘uCount’ as name and *bezierpatch_object.Udegree* +1 as value.

One remark is that all graphic transformations in FSPMs are applied to shapes. Particularly in the FSPM Apple project, the graphic transformations were applied to 3D shapes that represent real plant modules. The transformation matrix applied to an XEG node of shape type is to be interpreted relative to its parent node of shape type. Precisely, it is a matrix describing the transformation from the location of the reference point of the parent shape node to the location of the reference point of the child shape node. On the other hand, for the symmetrical shape types (e.g. *Cylinder*)

in the IMP3D library that are commonly used to graphically represent plant modules, the location of their reference point is normally a ‘starting location’ of such a shape object. A translation is by default applied to allow the current location to be updated to the ‘end location’ of the shape during graphical interpretation. On the other hand, the designed four different decomposition schemes follow the principle that the topological neighbors are also geometrical neighbors. Thus, if the translation component of the computed local transformation matrix is not zero, then it is from the starting location to the ending location of the child shape node. To avoid the translation to be applied twice when the XEG is imported into GroIMP, we set it as zero manually. That is reflected by the replacement of *d, h, l* by ‘0’ in the transformation matrix of the fifth step. Besides, we use a node of type *ShadedNull* to capture the local transformation matrix that excludes the translation component, and a node of type *Scale* with values 1 to capture the scales. The reason we do this instead of decomposing the local transformation matrix that excludes the translation component into a rotation and a scale is that the *BezierSurface* has two applied scales and a rotation, it makes no sense to decompose the local transformation matrix into two different scales and a rotation. Restoring the transformation using original types is optional, while ensuring the geometric correctness is mandatory. As long as the transformation matrix is correct, the information consistency is guaranteed.

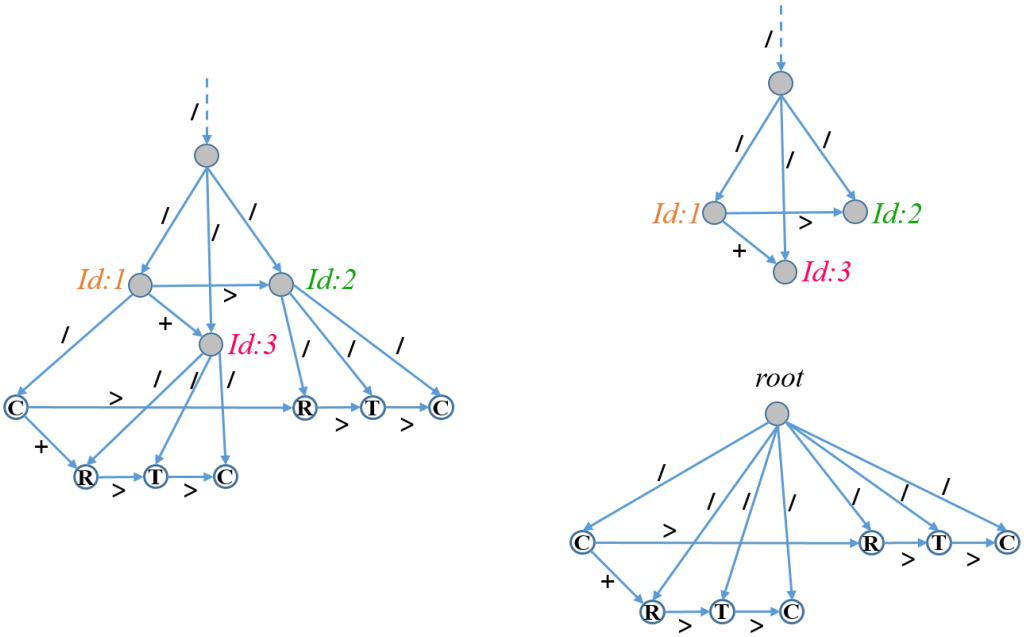
Another remark is that the essence of the node conversion from MTG to XEG is to find the type correspondence and to establish algorithms to convert properties for each correspondence. However, one type might have more than one signature with different combinations of properties that are equivalent with each other. It is therefore important to find a way to ensure all possible property sets have correspondences with appropriate property converting algorithms when the node transformation processes are developed for the integration of all FSPMs based on the same platforms. Here, the node transformation from MTG to XEG is designed for the integration of two specific FSPMs. Hence, in the sixth step, we simply

established an algorithm to convert the property set of an object of OpenAlea type actually used in MAppleT to the property set of an object of GroIMP type actually used in the GroIMP transport model.

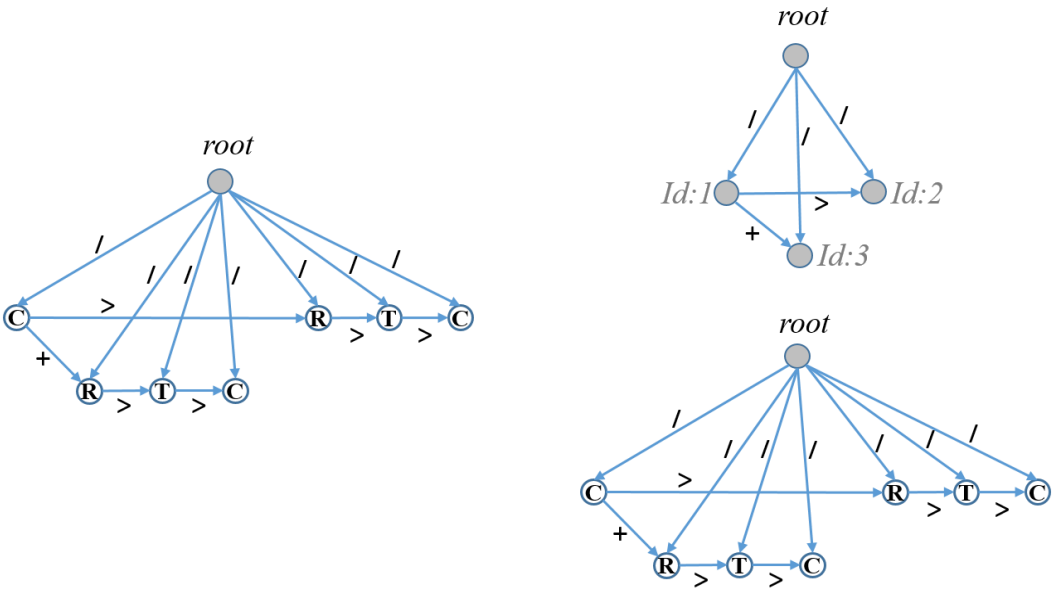
5.1.2.2 Algorithms for ETL processes from XEG to MTG

For the opposite direction of data flow at the client side, we designed a set of algorithms for nodes transforming accompanied by an algorithm for extracting FSP data from XEG and loading them to MTG. The former includes algorithms to convert nodes of most commonly used graphic types on GroIMP to graphic types on OpenAlea. The richness of the algorithms practically enables the transforming processes of the integration of all FSPMs based on OpenAlea and GroIMP. The latter includes an algorithm extracting the sub metamer scale and loading them to an object of PlantGL *Scene* type, and an algorithm extracting the other scales and loading them to an object of OpenAlea *MTG* type. The original objects of OpenAlea *MTG* type have a data field 'id' with data that links each node at metamer scale to its graphic properties, namely graphic objects managed by the object of *Scene* type, and we keep the id field of nodes at metamer scale. In this way, we guarantee the validation of separation of graphics to the object of *Scene* type when data have been exported back to MTG.

The extract processes for FSP data in XEG include a pre step to divide the XEG object into an XEG of geometrical structure and an XEG of non-geometrical structure. The former XEG is to restore the FSP data originating from the object of PlantGL *Scene* type. The latter XEG is to restore the FSP data originating from the object of OpenAlea *MTG* type. To allow the interface, the division of XEG has to consider FSP data originating from the simulation of FSPMs based on both platforms. We have designed a division scheme of two maps to allow the integration of all FSPMs (c.f. Figure 5.2).



A. Map for division of XEG encoding multiscale FSP data



B. Map for division of XEG encoding single scale FSP data

Figure 5.2 The division scheme of XEG

Following the map for multiscale FSP data (part A in the Figure 5.2), the nodes at the scale consisting of geometrical nodes are used to create the geometrical XEG with the intra scale topology. Moreover, the root of the geometrical XEG will decompose into all the nodes directly. The other part of the XEG will be used to create the non-geometrical XEG with topology unchanged.

Following the map for single scale FSP data (part B in the Figure 5.2), the geometrical XEG is created by a copy of the XEG to be divided. The non-geometrical XEG will have the finest scale with non-geometrical nodes corresponding to a group of nodes that consists of one shape node and all its ancestry transformation nodes until the nearest ancestry shape node in the topology of the XEG to be divided. Each non-geometrical node will have an id that corresponds to the order of applying BFS to the finest scale with a group being regarded as a node. The edges between the groups of nodes are copied to the non-geometrical XEG. In this way, all structural information is preserved when the divided XEGs are converted to the objects of *MTG* and *Scene* types. Meanwhile, the structure complies with the way to preserve transformations in OpenAlea and does not depend on any specific 'design of metamer' similar to MAppleT's four different kinds of graphical elements as apple tree modules, which are graphic properties of nodes at the metamer scale of the MTG. The essence of such division is topological upscaling.

For the geometrical XEG, the extract processes apply a Breadth-first search (BFS) algorithm. During the transversal, transforming processes for graphic transformations, shapes, and colors convert from GroIMP types to OpenAlea types. During the load processes, graphic transformations are applied to shapes and then the transformed shapes are added to the data structure of an object of PlantGL *Scene* type. We have divided the nodes within the geometrical XEG into different categories according to their data types and handled them differently.

XEG node of IMP3D type	MTG graphic objects of PlantGL type
Sphere(radius)	Sphere(radius), None
Box(length, width, height)	Translated(0, 0, z/2, Box(Vector3(length /2, width/2, height/2))) Matrix4.translation(Vector3(0, 0, height))
Cone(length, radius)	Cone(radius, height=length), Matrix4.translation(Vector3(0, 0, length))
Cylinder(length, radius)	Cylinder(radius, height=length) Matrix4.translation(Vector3(0, 0, length))
Frustum(length, baseRadius, topRadius)	Frustum(radius=baseRadius, height=length, taper=baseRadius/length) Matrix4.translation(Vector3(0, 0, length))
TextLabel(caption)	Text(caption), None
PointCloud(color, points, pointSize)	PointSet(pointList←points, colorList←color, width=pointSize), None
Parallelogram(length, width)	TriangleSet(pointList, indexList) pointList=[Vector3(0, 0, 0), Vector3(width, 0, 0), Vector3(width, 0, length), Vector3(0, 0, length)], indexList=[(0, 1, 2), (0, 2, 3)], None
Polygon(vertices)	TriangleSet(pointList←vertices, indexList←vertices), None
BezierSurface(uCount, data, dimension)	BezierPatch(ctrlPointList← (data, dimension)), None
NURBSCurve(ctrlpoints, dimension)	If dimension ==2: NurbsCurve2D(ctrlPointList← (ctrlpoints, dimension)), else: NurbsCurve(ctrlPointList← (ctrlpoints, dimension)), None
NURBSSurface(ctrlpoints, uSize, vSize, uDegree, vDegree, dimension)	NurbsPatch(matrixArray←(ctrlpoints, uSize, vSize, dimension), uDegree, vDegree), None

Table 5.1 Transform schemes for XEG nodes of GroIMP shape types

- XEG nodes of normal shape types, including *Sphere*, *TextLabel*, *PointCloud*, *Polygon*, *BezierSurface*, *NURBSCurve*, *NURBSSurface*, and XEG nodes of shape types that apply a default translation from starting location to ending location of the shapes, including *Box*, *Cone*, *Cylinder*, and *Frustum*. For such nodes, corresponding transforming processes take their properties to create an object of a PlantGL shape type and an object of PlantGL transformation type (i.e. a matrix instance) that captures the translation. For nodes of latter types, *None* is used for the transformation matrix. As Table 5.1 shows, *radius* and *length* of an object of IMP3D *Cylinder* type *oic* are taken to create an object of PlantGL *Cylinder* type *opc* and an object of PlantGL *Matrix4* type *opm*. The values of *oic*'s properties are respectively assigned to *opc*'s properties *radius* and *height*, and the *length* is applied in the creation of *opm* as well. In the table the symbol '→'is used to reflect transformations of properties that cannot be expressed in simple assignments.

- XEG nodes of normal graphic transformation types, including *Translate*, *Scale*, and *Rotate*, and XEG nodes of types for turtle commands that act as graphic transformations, including *M*, *RL*, *RU*, *RH*, *RV*, *RV0*, *RG*, *RD*, *RO*, *RP*, *RN*, *AdjustLU* as well. For such nodes, corresponding transforming processes take their properties to create a PlantGL transformation matrix.

- XEG nodes of types for a turtle command that act as shapes but take turtle states as parameters, including *F*, *F0*, and *M0*. For such nodes, corresponding transforming processes take their properties and relevant current turtle states (e.g. diameter, length) to create an object of a PlantGL shape type (i.e. *Cylinder*) or *None* (for nodes of type *M0*) and create a PlantGL transformation matrix.

- XEG nodes of types for turtle commands that modify states, including *V*, *Vl*, *VlAdd*, *VlMul*, *VAdd*, *VMul*, *L*, *Ll*, *LlAdd*, *LlMul*, *LAdd*, *LMul*, *D*, *Dl*,

DlAdd, *DlMul*, *DAdd*, *DMul*, *P*. For such nodes, corresponding transforming processes take their properties to modify relevant turtle states. Particularly for *ShadedNull*, which is not a turtle command, processes modify turtle states and create a PlantGL transformation matrix.

To allow the transform processes for the geometrical XEG, an object to store current turtle states, an object of PlantGL Scene type, and a map to store the current parent nodes and their global transformation matrix (iterative products of local transformation matrices produced when applying the BFS) are needed.

In detail, the ETL processes start from the root of the geometrical XEG, which is not typed, thus no transform process is carried out so that nothing is produced for loading to the object of PlantGL Scene type, an entry (root:None) is stored in the map with *None* as the global transformation matrix GM of the root. Following the BFS, children nodes of the nodes traversed in the previous step will be traversed as nodes of the current generation. Corresponding transform processes are carried out according to the schemes defined in different catalogs. If the transforming result of an XEG node of the current generation **cnd** includes a transformation matrix M, its parent node's global transformation matrix multiplied by M is stored in the map as the current node's global transformation matrix GM. If the result has also a shape, then the GM is applied to the shape, and the transformed shape is merged in the object of PlantGL *Scene* type. Before the next BFS step, an entry of the form (**cnd**, GM) is stored in the map. Note that when the operand of a matrix multiplication is *None*, it is replaced by an identity matrix.

As L-py does not include the turtle commands as MTG elements, the transform processes of turtle commands are actually for the execution of the models based on GroIMP from OpenAlea, which is a special use case of the integrative interface. To allow a relatively comprehensive usage of turtle commands, we have implemented the transform processes for most commonly used types of turtle commands and a turtle state object with all interaction algorithms/patterns originating from GroIMP.

One remark concerns the transform processes for nodes of shape types. During a process, an XEG node is processed according to its type originating from GroIMP. For most type correspondences, the processes are about to convert the properties to suit the corresponding OpenAlea type, i.e. a type available in PlantGL. For the types Parallelogram and Polygon, there are no graphically equivalent types in PlantGL, thus we have to use alternative types. In our implementation, we used the TriangleSet type in PlantGL, and the transform processes for nodes of the two types are based on triangulation algorithms correspondingly. For the type Parallelogram, we designed a simple algorithm that divides a parallelogram into two head-to-tail congruent triangles. For the type Polygon, which is a general case, we use a general algorithm, i.e. the Delaunay triangulation algorithm[153].

Another remark is about the problem that one correspondence of types can have several equivalent property list/signature correspondences. Unlike the previous transform processes from MTG to XEG that consider only specific signature correspondences, the transform processes in this part are supposed to be applicable for all FSPMs based on GroIMP and OpenAlea. Therefore, we have designed general algorithms for each type correspondence and implemented them with a specially designed paradigm available in Python. It combines the **args* and ***kwargs* Python syntax for defining methods with an indefinite number of parameters and the *object.__getattribute__(self, name)* [154] for calling the defined methods with a given node type.

Besides, it is clear that the algorithm might not restore exactly the transformation with original types such as *Oriented*, *Translated* but with the transformation matrix type *Matrix4*. The reason is the same as that given in the first remark in the last section. Each transform scheme for nodes of types *Box*, *Cone*, *Cylinder*, *Frustum* creates not only an object of PlantGL shape type but also an object of PlantGL transformation matrix type *Matrix4* corresponding to the default translation from starting to ending location of the shape. This effectively ensures the geometry

correctness of the transform process from XEG to MTG. To ensure the correctness of geometrical transformations, the topology at sub metamer scale has been restored by decomposing the transformation matrix of PlantGL *Matrix4* type to the pattern of applying the transformation to our three types of shapes (c.f. section 5.1.2.1). The order of the graphic objects to be restored to the object of PlantGL *Scene* type has been particularly taken care of (c.f. Figure 5.1).

For the XEG of multiscale structures, the extract processes need to be driven by a spanning algorithm because at each scale the RGG graph has a general graph structure while the MTG has a tree structure. The algorithm contains a step to locate the root of a scale. It starts from the graph root, and then proceeds to the root of the next finer scale. When such a root is located, a process to carry out a BFS (Depth-First-Search, DFS could also be used) starts. During this process, a tree structure using the MTG API is constructed, i.e. a load process while extracting. For each BFS step, we create corresponding MTG nodes for XEG nodes just traversed and we create MTG edges for existing edges from the parent node to the current traversed nodes in the XEG. We create also MTG edges for existing outgoing edges of decomposition type of XEG nodes just traversed.

It should be emphasized that the geometrical XEG has also a general graph structure. We do not take the topology into this structure, so the algorithm is simply the BFS (without creating edges). Besides, as there was no demand in the project, we have implemented only partially for the map B of the division scheme of XEG shown in part B of Figure 5.2. The implementation currently allows the geometrical XEG to be produced, but not the non-geometrical XEG.

Another remark is that the ETL processes for the geometrical and non-geometrical XEG work cooperatively. This is mainly required by the geometrical XEGs generated by the simulations of FSPMs based on GroIMP that might have nodes of extended types. To handle such a situation, the functional properties obtained through a type extension are added to the nodes at metamer scale of the

MTG object produced by the ETL processes for the non-geometrical XEG. Consequently, the ETL processes for the non-geometrical XEG are executed earlier than the ETL processes for the geometrical XEG.

5.2 Design and implementation of the component *ServerSideInterface*

In our designed component architecture of the FSPM integrative interface, the component *ServerSideInterface* consists of six components, which can be divided into a group for communication and interaction between FSPMs and a group for ETL processing. The former group includes the component *Client* and the component *Message* as implementation of the FSPM integrative protocol for receiving & deconstructing the integrative simulation request from the client and constructing & sending the simulated response to the client. As a part of the component *ServerSideInterface*, the group should also include the component *RetroactionChecker* and the component *SeverFSPMRunner* for calling the main method of the server FSPM and the coordination of retroaction of FSPMs. The latter group includes the component *Graph* and the component *GraphConverter* for FSP graph conversion between XEG and RGG graph. These are the main parts that correspond to the ETL pipeline and have been the focus of the introduction.

5.2.1 The communication group at server side

In the communication group of the component *ServerSideInterface*, the component *Server* is for receiving and responding the HTTP messages based on the specification of the HTTP protocol. The component *Message* is for request message destruction and response message construction based on the specification of the FSPM integrative RPC protocol. In our specific case, the non-retroactive use case is just for the user to run FSPMs based on GroIMP through the OpenAlea platform,

the component *RetroactiveChecker* is thus just an if-else statement as a part of the component *Graph*. For the *ServerFSPMRunner*, we have implemented a module using APIs available on GroIMP to get the target FSPM managed by the editor JEdit, then to compile the FSPM and run its 'main method'. The focus of this group is thus on the components *Client* and *Message*.

In detail, we have implemented the Master-Slave Pattern to allow simultaneous requests from multiple clients. The implementation includes a master module as http request listener and a slave module as http message handler. These correspond to the components *Client* and *Message* respectively. When a request is received, the master instantiates a slave thread to handle the request, and then resumes listening. In the meantime, the slave continues its communication with the client. The implementation is similar to the implementation of the communication group at client side, but the modules for the components *Client* and *Message* at client side do not have a master-slave relationship, i.e. an instance of a module for the component *Client* cannot create an instance of a module for the component *Message*. This way ensures sequential execution of FSPMs. By applying the implementation at server side, the master modules create an HTTP Service bind with a created slave thread and the Socket (IP address + Port number). The slave thread takes an initialized graph from the GroIMP current workbench and the extracted FSPM integrative protocol members 'model', 'main_method', 'graph', 'time', 'restorative' and 'id' to execute the module for the component *ServerFSPMRunner* and to reply the generated response message to the client through the bind Socket. The component *ServerFSPMRunner* uses the Java Reflection mechanism applied in GroIMP to get the corresponding method of the compiled model in the current workbench and iteratively run it using the graph just converted from the XEG as initial data.

5.2.2 The ETL group at server side

In the ETL group of the component *ServerSideInterface*, the components *Graph* and *GraphConverter* are for ETL processes between XEG and RGG graph. Among the three processes of ETL, the extract process is driven by the BFS graph-traversing algorithm and the load process is carried out according to the extracted FSP data. Similar to the client side, the extract process is based on an implemented XEG library in Python and the focus of extract and load processes is to ensure the correctness of the graph topology. As the transform processes for XEG to RGG graph have been included to the client side and there is no transform process for RGG graph to XEG at server side, there is actually no transform process on the server side. However, because of the specific technical setting of the RGG three-part-graph, we have modified the topology of the XEG to fit this setting when the extracted topology from the XEG is loaded to the RGG three-part-graph. Similar to the server side, data of each data field of FSP graphs need to be extracted and loaded according to the modeling platforms on which the two FSPMs are based, namely GroIMP and OpenAlea. This means the processes shall be applicable for the integration of other FSPMs based on the two modeling platforms. Beside the topology and geometry, other kinds of data fields such as colors have been considered as well.

To allow the server side to process the request of the RPC call from the client side, we have designed an algorithm to extract FSP data from an XEG and load it into an RGG graph. To allow the server side to response the RPC call, we have designed an algorithm to extract FSP data from an RGG graph (generated by GroIMP) and load it into an XEG. The implementation of the ETL group at client side has its focus mainly on the load processes of topology and nodes of the XEG to the RGG graph and vice versa. In detail, the RGG three-part graph is a pseudo multiscale graph as a general version of MTG with a general graph at each scale. Graph operations, such as graph query, can be performed as it is a multiscale graph.

However, its actual ‘single’ or multi-scale topology for one or multiple plants does not comply with standard scaled topology for one or multiple plants because of GroIMP’s graph setting and rendering mechanism. In fact, the RGG grammars expect a connection of the graph root to direct neighboring nodes not with decomposition edges, but with zero or more branch edges and optionally a successor edge. One plant is topologically connected with others with branch edges. When the multiscale concept is introduced, a type graph and a scale graph is added to the original graph to form a three-part graph to allow different scales to be rendered differently. The root of each scale is connected not only with a node of the next-coarser scale by a decomposition edge but also with the graph root by a branch edge. This setting allows the rendering paths of a RGG three-part graph to be activated and deactivated easily by changing the type of edges from the graph root to the roots of the scales. Consequently, for loading the XEG topology to the RGG three-part graph and vice versa, it is necessary to distinguish the case of ‘single’ scale from multiscale. Besides, the RGG graph root can be connected to at most a successor edge and one or more branch edges according to the specific code of a GroIMP model because the RGG graph structure is the result of compiling of XL code, while in our project, the involved FSP data is generated by MAppleT.

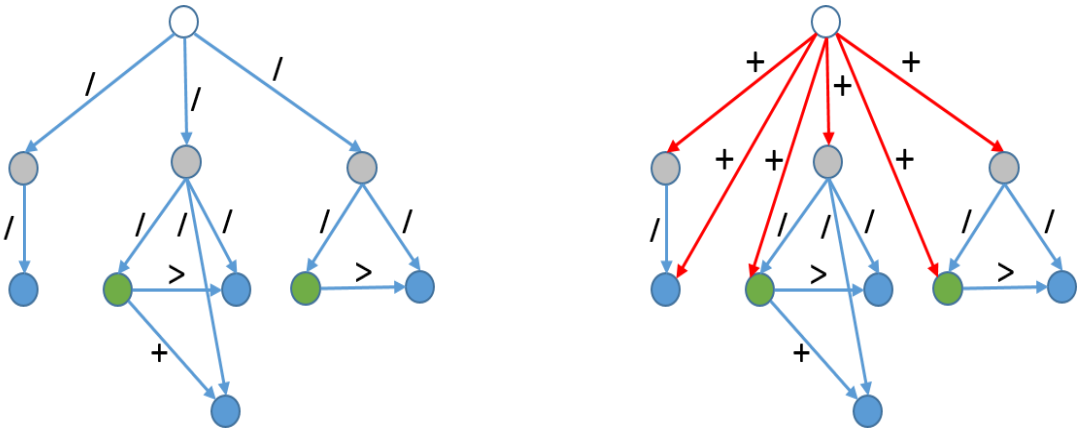


Figure 5.3 Topological map between XEG with multiscale FSP data (left) and RGG graph (right)

Therefore, when the FSP data encoded in the XEG is extracted and loaded into the RGG graph, how the RGG graph root is connected is indeterminate. To address this issue, we have designed a simplified scheme to map the topology within XEG and RGG graph. Figure 5.3 shows the map for multiscale topology. In the topology of an XEG with multiscale FSP data, there are only decomposition edges between nodes at different scales, while in the topology of a transformed multiscale RGG graph, there are also branch edges between the graph root and the root of each scale. Particularly for the nodes at the scale just finer than the graph root (i.e. whole stand or tree scale), the edges connecting the graph root to them are practically a replacement of decomposition edges by branch edges. Figure 5.4 shows the map for 'single' scale topology. For a 'single' scale XEG, the one or multiple plant cases are processed differently. The 'single' scale RGG graph of multiple plants keeps

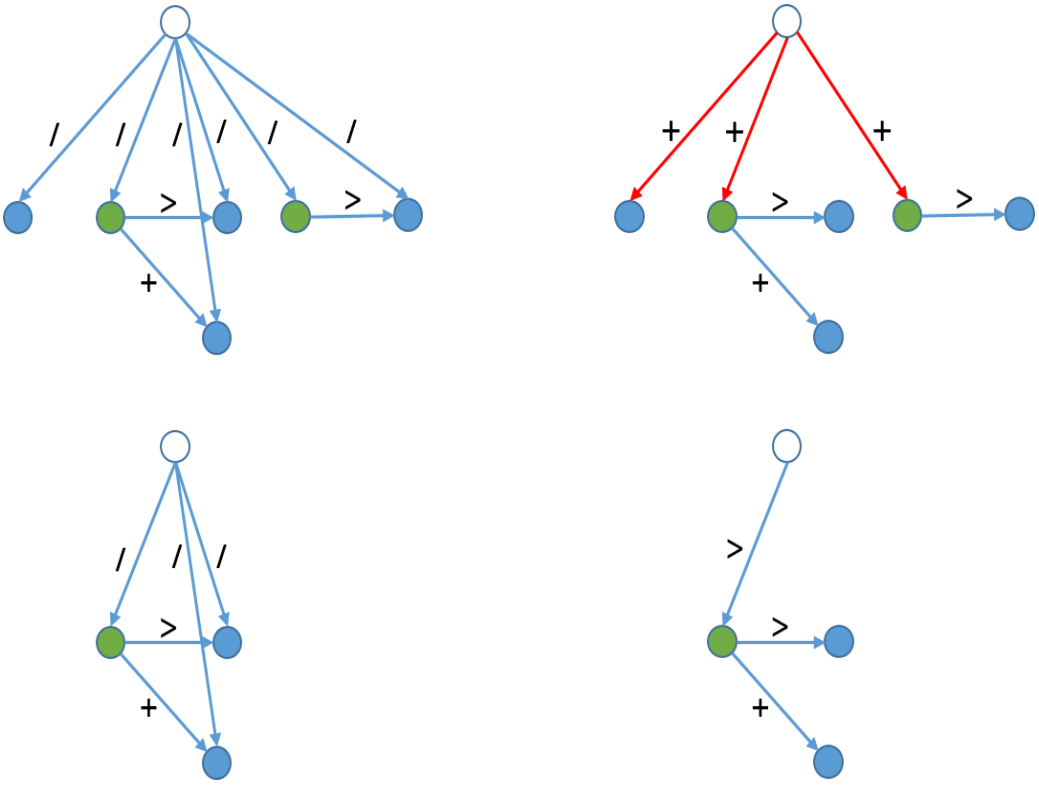


Figure 5.4 Topological map between 'single' scale XEG (left) and RGG graph (right)

the same correspondence as for the multiscale case, while in the RGG graph for one plant, the graph root is connected by a successor edge. Maps in the two figures are bidirectional, which means they describe the ETL pipeline for directions both from XEG to RGG graph and vice versa. Note that the designed scheme is about to map the topology of FSP data originating from MAppleT. The FSP data originating from a GroIMP model simulation can surely be represented with a topology that contains both successor and branch edges, but when they are transformed through the ETL pipeline, the resulting topology will comply with the designed scheme. For example, a typical RGG graph for a single plant with a 'single' scale has the form shown in the bottom right of Figure 5.4. But if the graph root is connected by a branch edge, the bottom left will be the topology of the resulting XEG, and when the XEG is converted back to a RGG graph, the bottom right will be the topology of the resulting RGG graph, which is different from the original topology. The reason to do so is that the RGG graph is a general graph allowing an arbitrary configuration yet we need a definite scheme to bridge the gap of topological difference. Thus, we have designed the scheme that ensures the 'unchanged' mapping of topology only for typical usage.

Beside the topological correspondence for the load processes, we have implemented a mechanism for the property correspondence. The nodes at metamer and above metamer scales are nodes of non-geometrical types with various properties. There is a mechanism in GroIMP to declare a *Module* as a new type that extends a type existing in GroIMP so that certain new properties can be added, but there are no data types that directly allow indefinite properties. We have implemented a new type called PropertyNode with a data field of *java.util.List* type. Together with another type called Property, this effectively enables the loading of nodes of non-geometrical types at metamer and above metamer scales. On the other hand, as we want the ETL group to be applicable for all FSPMs based on the two platforms, our implementation of the load process allows RGG graph nodes of extended types which are obtained from user-declared *Modules* to be expressed in

the XEG (c.f. Figure 4.7), and also XEG nodes with such extended types to be expressed in the RGG graph. The load processes also provide a solution for the issue of multiple signatures for the same type: We combine a java list and the java reflection mechanism [155] to allow various properties to be held temporally and to be assigned to an object (as RGG graph node) as properties.

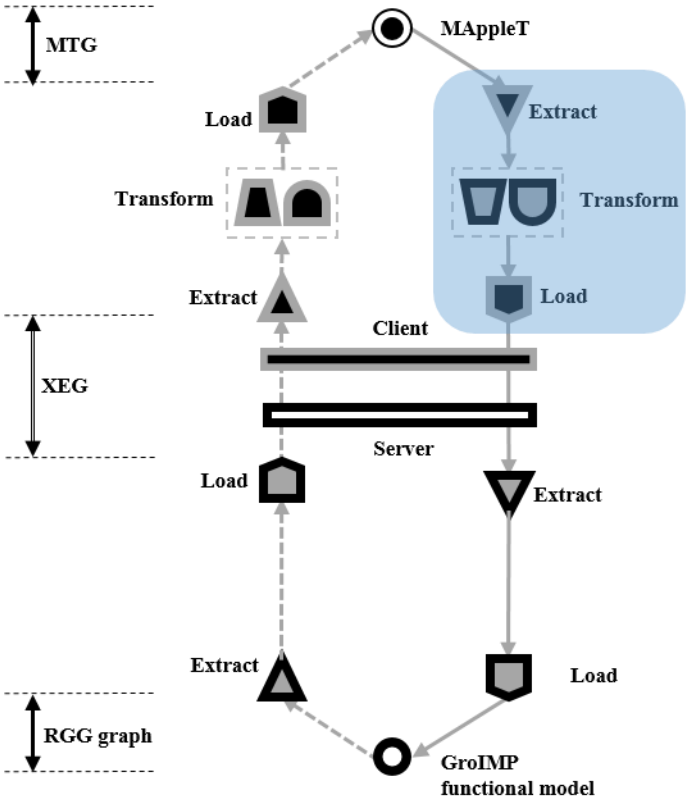


Figure 5.5 The instance architecture of the implemented interface for the integration of target FSPMs

5.3 Distinguishing features of the interface

One feature of the interface is the instance of architecture of the implemented interface, as shown in Figure 5.5. The ETL processes from MTG to XEG at the

client side have a specific implementation that is only applicable for MAppleT, all the other parts of the interface are applicable for all FSPMs based on OpenAlea/GroIMP.

Another feature is that the ETL groups at server side are applicable not only for the integration of the target FSPMs, but also for the invocation of any GroIMP model from the OpenAlea platform. The role of the integrative FSPM can be shifted from one model to the other. In other words, FSPMs based on GroIMP can be client/source and FSPMs based on OpenAlea can be server/target when the corresponding server and client are further provided.

Chapter 6

APPLICATIONS AND ENHANCEMENTS

In this chapter, we introduce the applications of the interface and the enhancements realized for GroIMP and the interface. The applications include geometrical upscaling using an XEG with multiscale FSP data converted from an MTG and the integrative simulation of the target FSPMs of the FSPM Apple project. The enhancements mainly result from the development of graph query commands as an addition to the vocabulary of the language XL.

6.1 Geometrical upscaling

After introducing the multiscale concept to the RGG graph, the original single part graph has been replaced by the three-part graph. The added parts, i.e. the scale and type graphs clarify the relations between different scales and data types used in a particular scale [106]. Together with its special multiscale topology mentioned in the last chapter, rendering of a particular scale becomes possible. On the other hand, there is a concept called Level of Detail (LOD) [156] in the field of computer graphics, which is a kind of technique of interactive computer graphics that attempts to compromise complexity and performance by regulating the amount of detail used to visually represent the virtual world. The basic idea of LOD is to increase the

graphical performance by rendering a scene with a less complex graphical representation that reduces details for small, distant, or unimportant portions. Many LOD algorithms have been introduced since the last decade. For us, this approach has an additional value: FSPMs in principle use graphics to represent plants at organ level. With the introduction of the multiscale concept, the graphical representation of plants at organ level is supposed to be distributed to different spatial scales so that plant functions can be simulated at different spatial resolutions. This can bring advantages such as simplification of FSPMs using production rules at coarse scales and higher performance of the computation of functional properties associated with less complex shapes presenting plant modules. However, there are not much algorithms established to achieve such purpose. In our project, we have the XEG converted from the MTG which is produced by a MAppleT simulation that encodes multiscale plant structures with geometrical objects at the finest scale. We thus take this opportunity to try to develop LOD algorithms to allow the plant functions to be simulated at different scales, which we call geometrical upscaling.

The basis of our algorithms is the bounding volume concept, namely using a closed but simpler volume that completely contains the union of a set of geometrical objects to improve graphical performance. The common bounding volumes include bounding box, bounding sphere and convex hull. We have developed two algorithms of geometrical upscaling using bounding box and convex hull. Applying the algorithms, a bounding volume of geometrical objects (also nodes) at a finer scale is computed to represent a node at a coarse scale that directly or indirectly decomposes into the nodes at the finer scale. In detail, the first step is to perform a RGG graph query to get all nodes at a coarse scale. These nodes in our case are non-geometrical nodes and are of PropertyNode type after being loaded to the RGG graph. The nodes are retrieved by using RGG query syntax. For such a node at a coarse scale, the nodes into which it decomposes at the sub metamer scale of the XEG are used to compute a bounding box or a convex hull of a type that extends both PropertyNode and relevant geometrical types (here we have used MeshNode

as the base type.). Then all properties of a node at the coarse scale are duplicated to the computed bounding box or convex hull. Finally, the corresponding volume replaces the node at coarse scale through a graph rewriting. In addition to the main steps, a pre step that creates the type and scale structure is necessary to allow the scales with different geometry to be visualized in an interactive manner.

The key of the algorithms are the methods to compute the bounding box and convex hull of a set of geometrical objects of Cylinder, Sphere and BezierSurface type. The idea of the methods is to get the extreme points of the geometrical objects of such types at their default positions in the coordinates systems. For the Cylinder, its default position is defined as the basal circle lying in a plane spanned by the X and Y-axes, with its center as coordinate origin. Thus, the defined extreme points are expressed using its parameters, the length *l* of the cylinder and radius *r* of the basal circle, as follows:

$$(r, r, 0), (-r, r, 0), (-r, -r, 0), (r, -r, 0),$$

$$(r, r, l), (-r, r, l), (-r, -r, l), (r, -r, l)$$

For the Sphere, its default position is defined as the center of the sphere being in the origin of the coordinates. Thus, the defined extreme points are expressed using its parameter, the radius *r* of the sphere, as follows:

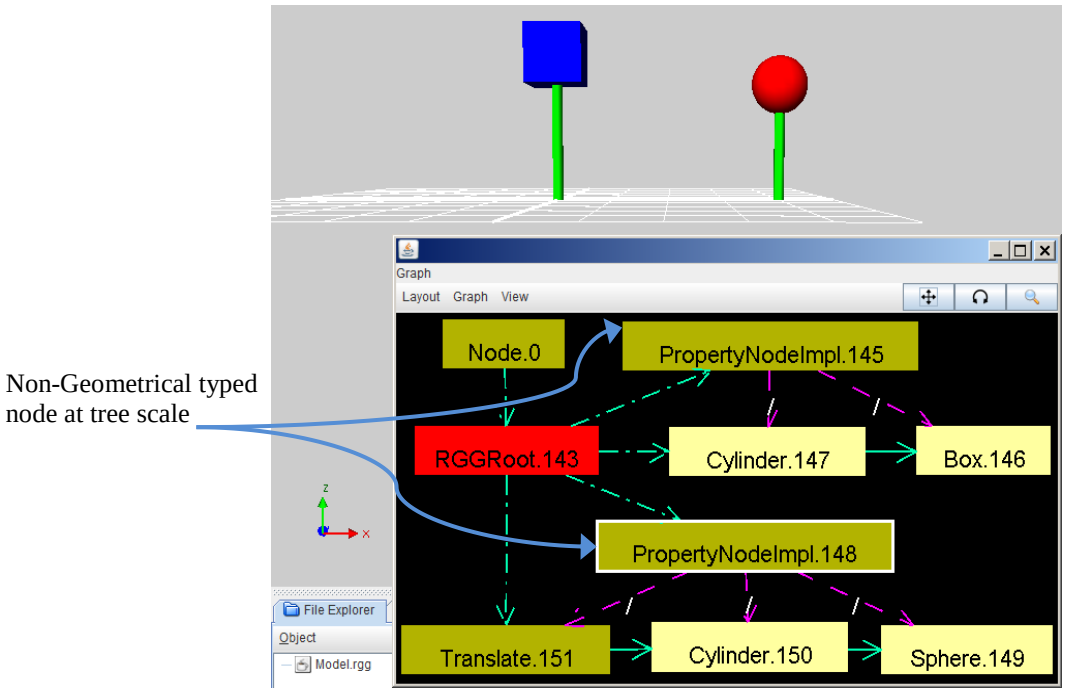
$$(r, r, r), (r, -r, r), (-r, r, r), (-r, -r, r),$$

$$(r, r, -r), (r, -r, -r), (-r, r, -r), (-r, -r, -r)$$

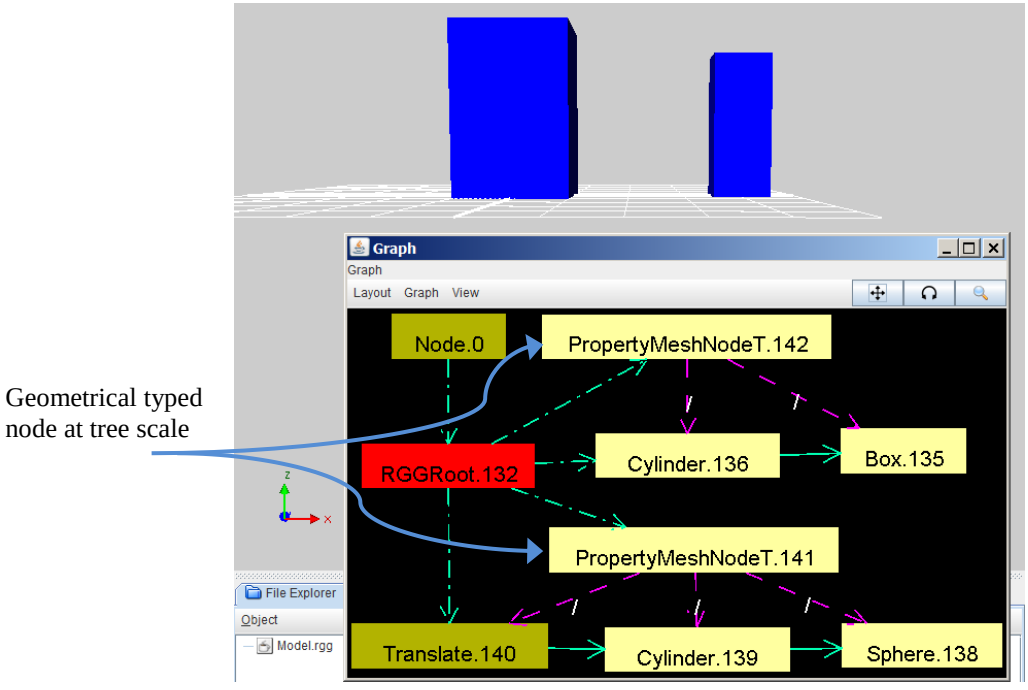
For the BezierSurface, its control points are directly used as its extreme points, which are constructed using its parameters data (i.e. a float array of coordinate components of all control points) and dimension. This leads to a bounding volume that is just an approximation of the minimal bounding volume.

For the algorithm of computing the bounding box and convex hull, we use in both cases a three dimensional implementation of the Quickhull algorithm [157], which computes the convex hull of a set of 3D points. We did it in this way for the following reasons. On one hand, a box can be constructed as a convex hull of eight extreme points. On the other hand, the issue of default translation where using directly the IMP3D type Box for computing the bounding box is avoided. One remark is that the computed bounding box is axis aligned as we compute the extreme points from the transformed shapes (i.e. shapes with global coordinates).

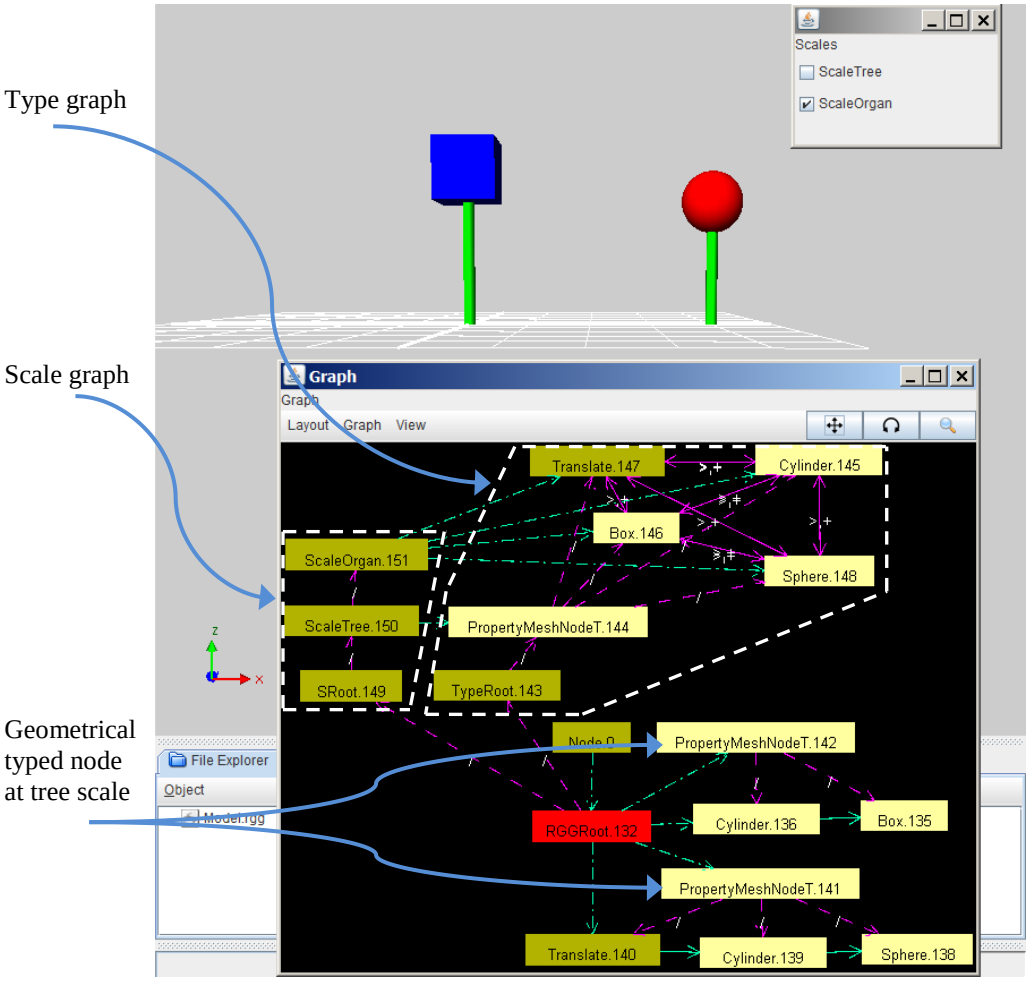
We have done an experiment with an XEG of made-up structure of two schematic plants with an organ and a tree scale to test the geometrical upscaling and the switch of scales with type and scale graphs. This effectively validates the extract and load processes at server side from XEG to RGG graph for multi-scales/plants cases. In Figure 6.1, part A shows the RGG graph extracted and loaded from XEG. Part B shows the result of geometrical upscaling applied to the RGG graph. The non-geometrical node of 'PropertyNodeImpl' type at coarse scale has been replaced by the geometrical node of 'PropertyMeshNodeT' type, which is the bounding box computed from the corresponding organs at finer scale. Part C shows the result of adding scale and type graphs after the geometrical upscaling. Those two graphs makes it is possible to activate and deactivate the rendering of a scale (i.e. to switch the scales).



A. An XEG of two plants with two scales converted into an RGG graph



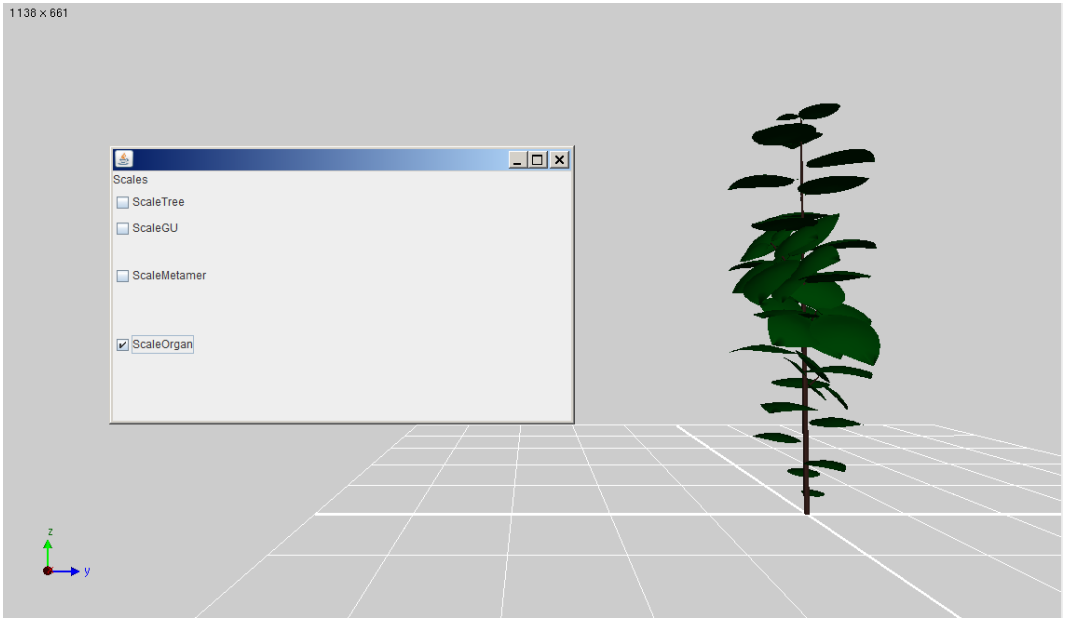
B. Geometrical upscaling of the RGG graph converted from XEG



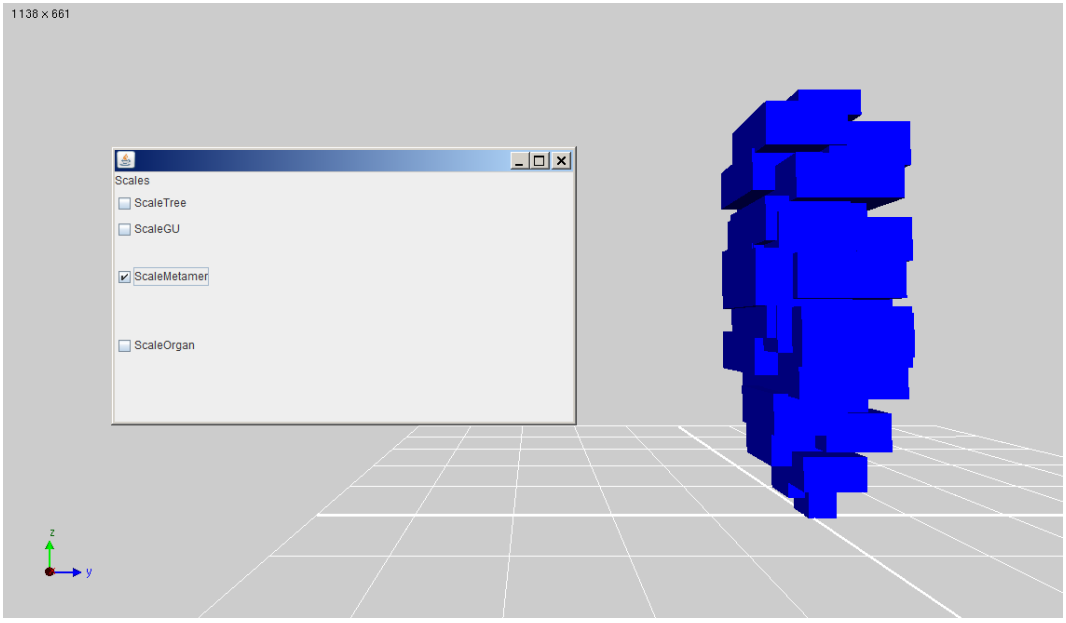
C. Geometrically upscaled RGG three-part graph after switching off the tree scale

Figure 6.1 Geometrical upscaling with bounding box for multiple plants at two scales from which an interactive choice is possible by the panel in the upper-right corner.

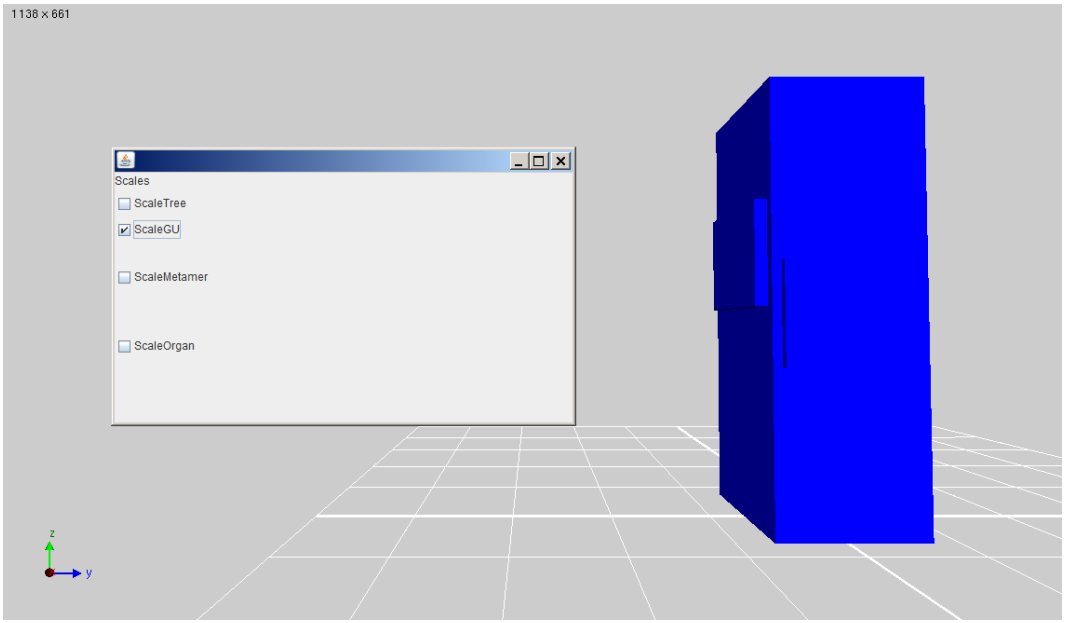
We then geometrically upscaled an XEG converted from an MTG (c.f. XL code in section 7.3), which encodes a plant having an additional organ scale and original metamer, growth unit, tree scales with ‘leaf’ and ‘internode’ metamers, using both bounding box (c.f. Figure 6.2) and convex hull (c.f. Figure 6.3).



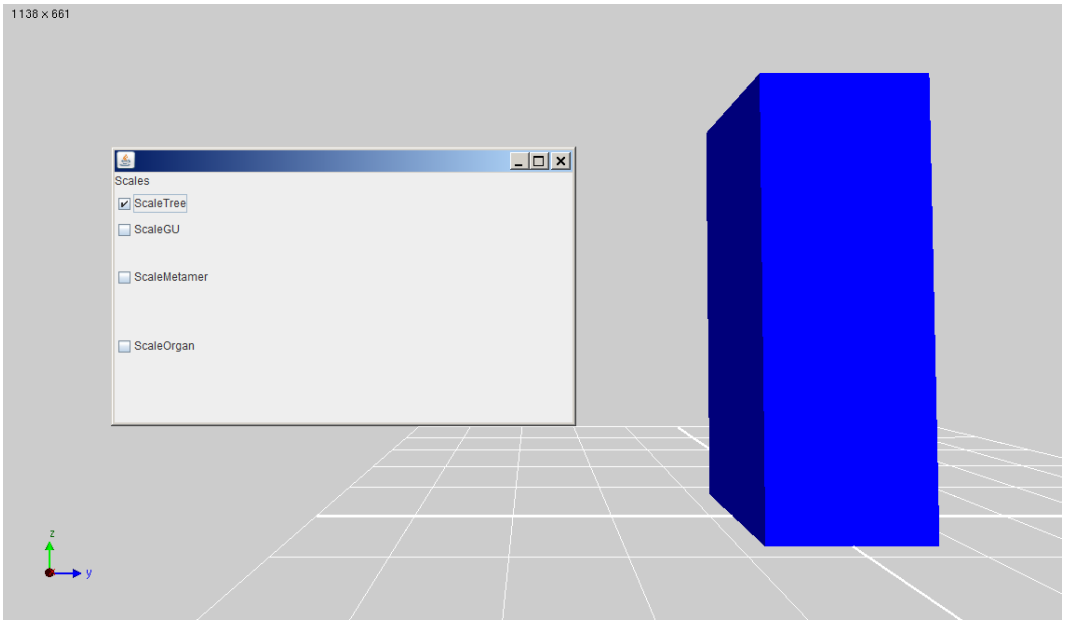
A. Original Geometry from the RGG graph at organ scale



B. Geometry upscaled from organ to metamer scale

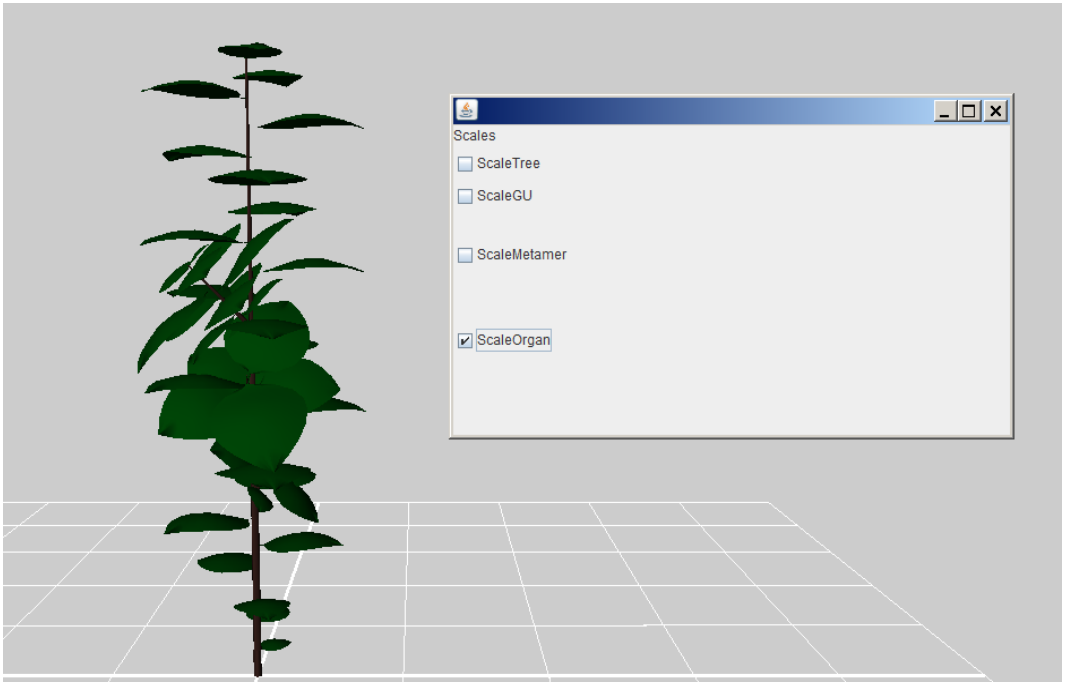


C. Geometry upscaled from organ to growth unit (GU) scale

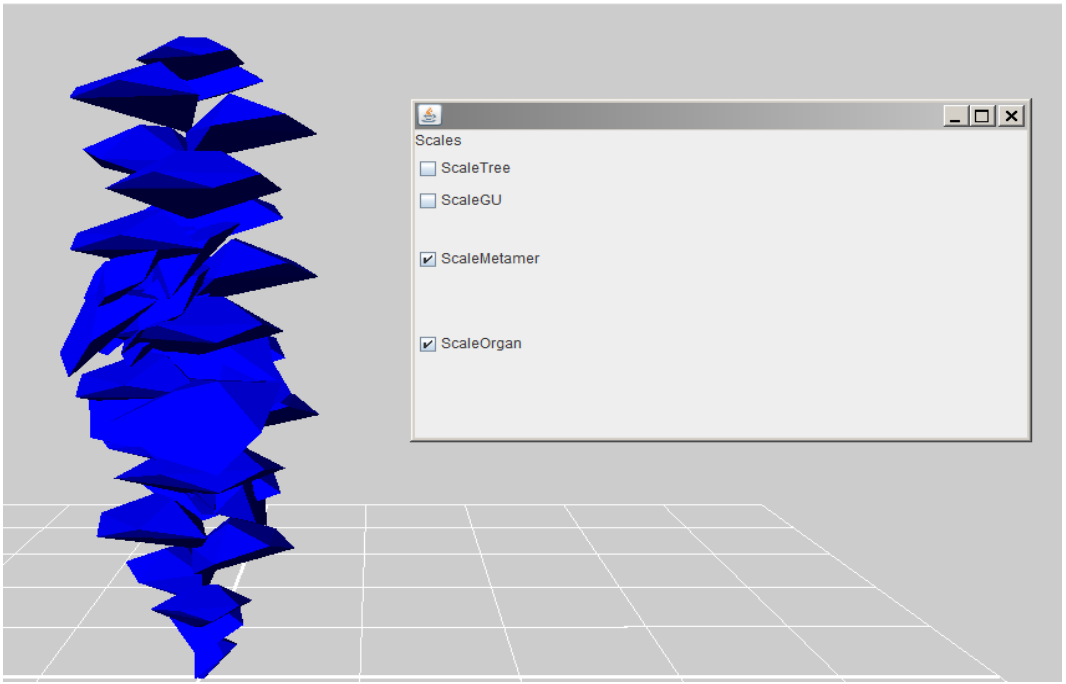


D. Geometry upscaled from organ to tree scale

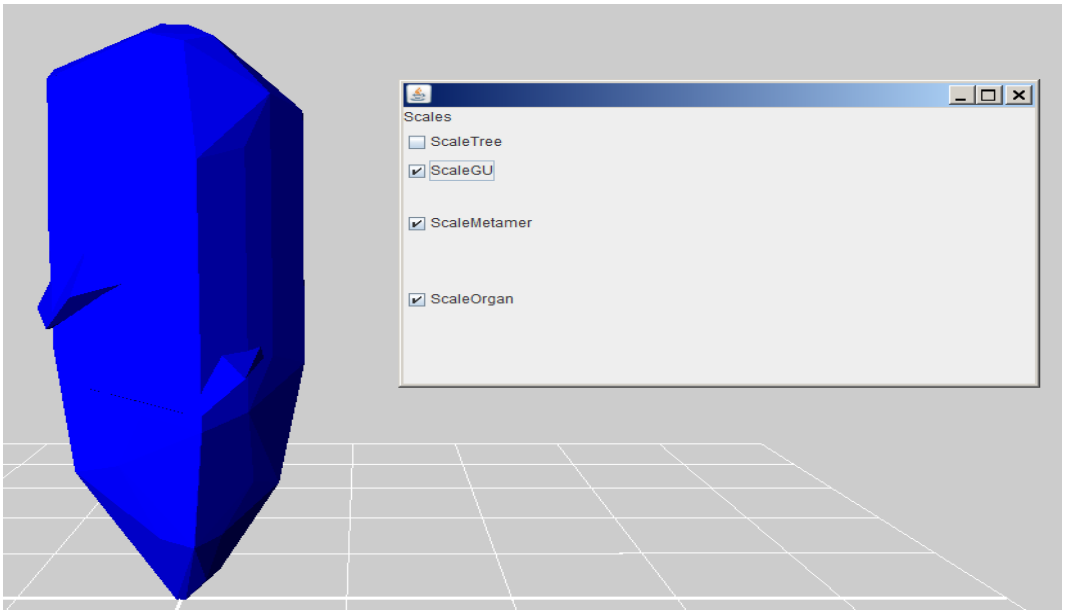
Figure 6.2 Geometrical upscaling with axis-aligned bounding box. The data originally encoded in the MTG are loaded into the RGG graph with their original geometry at the additional organ scale (A) and geometries upscaled to metamer scale (B), growth unit scale (C), and tree scale (D).



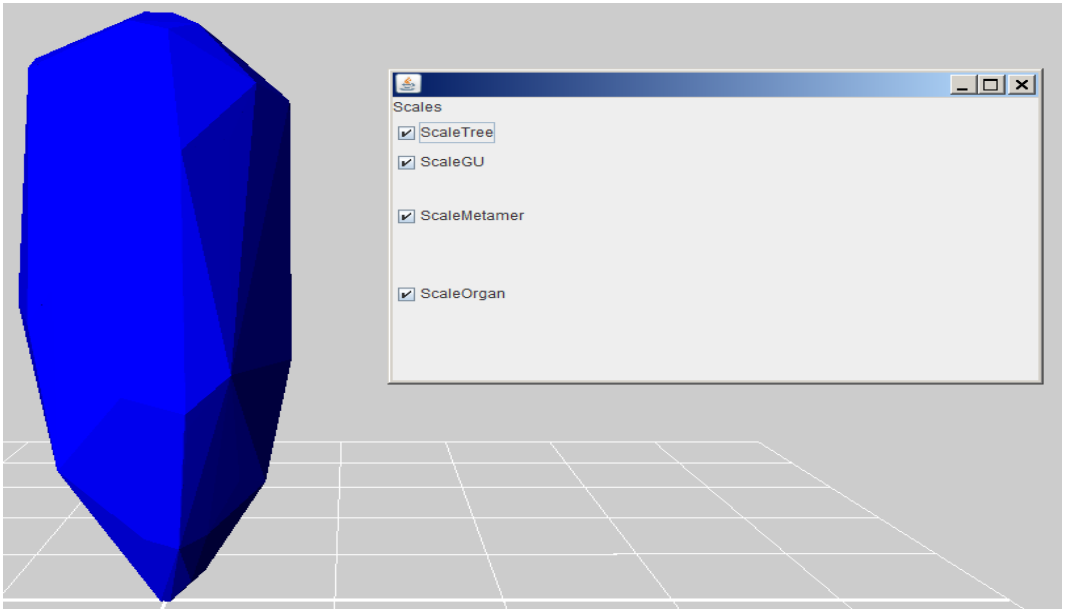
A. Original geometry from the RGG graph at organ scale



B. Geometry upscaled from organ to metamer scale



C. Geometry upscaled from organ to growth unit (GU) scale scale



D. Geometry upscaled from organ to tree scale

Figure 6.3 Geometrical upscaling with convex hull. The data originally encoded in the MTG are loaded into the RGG graph with their original geometry at the additional organ scale (A) and geometries upscaled to metamer scale (B), growth unit scale (C), and tree scale (D).

6.2 The integration of different FSPMs using the interface

After comprehensive introduction of the integrative middleware, i.e. the interface, we move to the application. Before we integrated the target FSPMs, we have performed several experiments which exhibit the practical usage of the interface and validate its implementations.

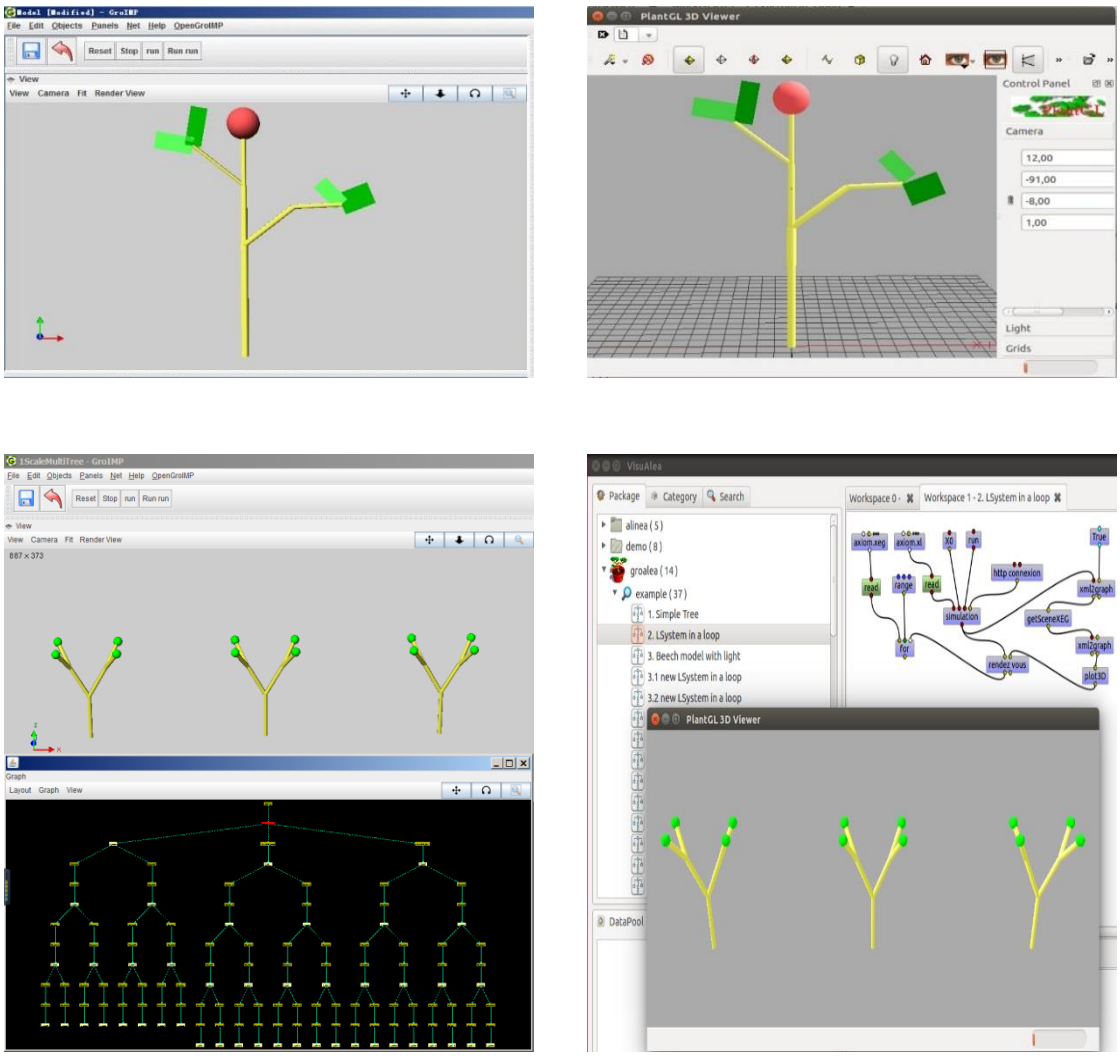
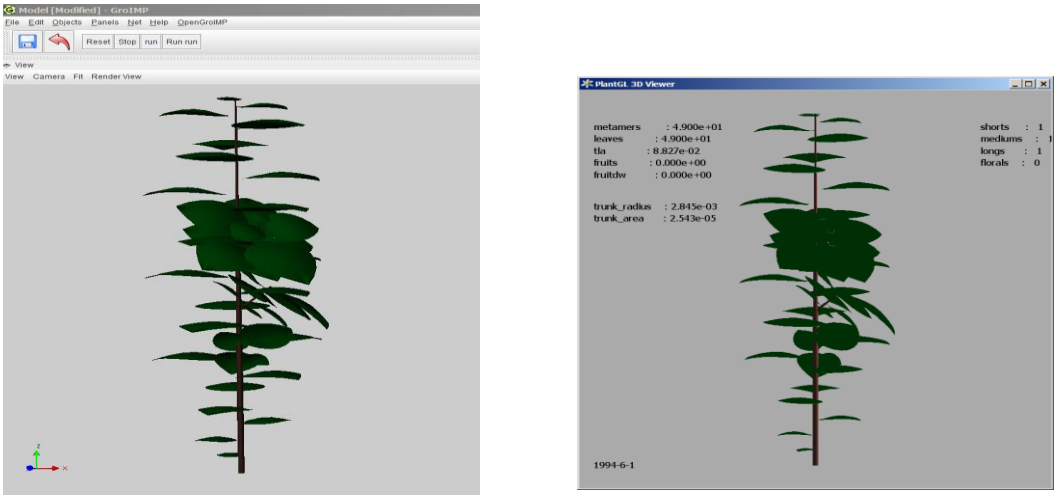
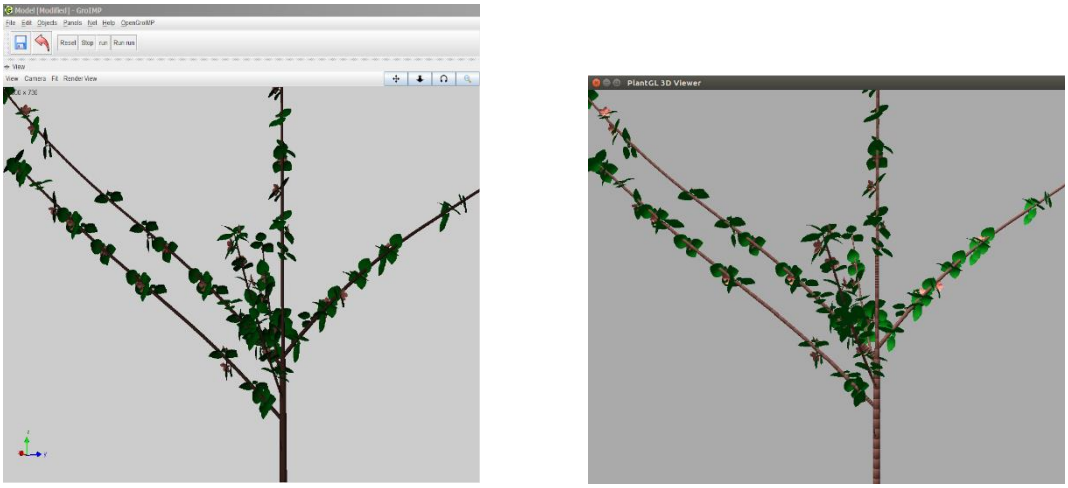


Figure 6.4 The identical results of the same GroIMP model directly run on GroIMP (left) and invoked from OpenAlea through a FSPM integrative RPC call (right).

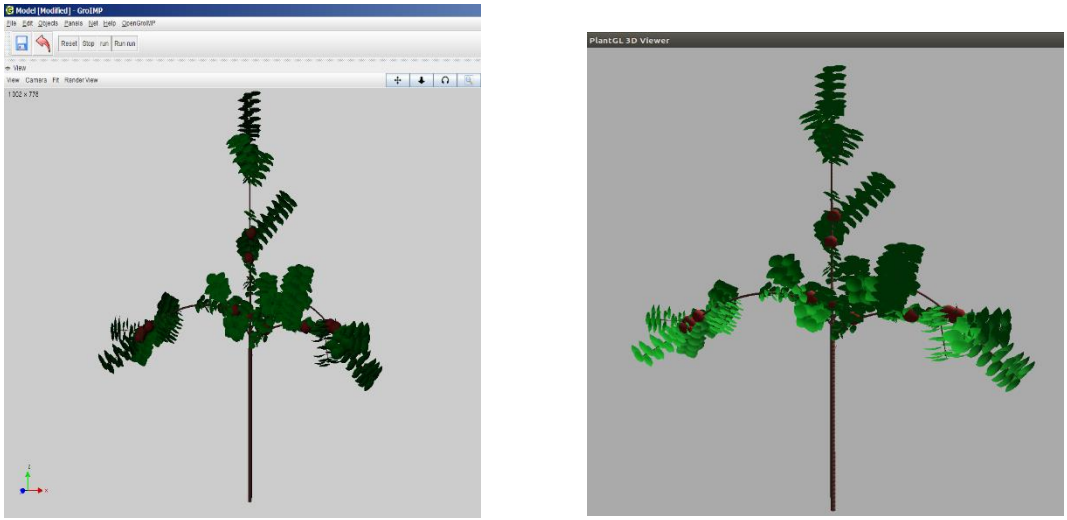
We have firstly tried an experiment to test the interface by applying a simulation of a GroIMP model at the OpenAlea side, that means to just make a FSPM integrative RPC call by directly providing all the request members of message body including the model (i.e. XL code, it is possible also to have the it at server side). Figure 6.4 shows the graphic views of the simulation results of a GroIMP model. The right panel shows the graphic view of the simulation 'executed' at the OpenAlea side by making a FSPM integrative RPC call. The graphic view is identical with the graphic view of the simulation executed directly at the GroIMP side. Notice that an object of PlantGL TriangleSet type consists of two objects of PlantGL Triangle type and represents an object of Parallelogram type of IMP3D. It is also noticeable that the message structure members 'graph' and 'model' here are single scaled. Consequently, the experiment validates the implementations for the communication group at both client and server side, the ETL group at both sides, especially the transform process at client side including turtle commands and triangulation of Parallelogram, the extract and load processes at both sides for a topology of a 'single' scaled model and multiple plants.



A. FSP data representing a small apple tree



B. FSP data representing an apple tree with flowers



C. FSP data representing an apple tree with fruits

Figure 6.5 FSP data in RGG graph (left)/MTG (right) after ETL processed

We tried a second experiment to test the interface by applying ETL processes to three MTGs generated by MAppleT simulations that encode a small apple tree with only internodes and leaves, a medium apple tree including flowers, and a large apple tree with apple fruit. Figure 6.5 shows the three virtual apple trees encoded in RGG

graph and MTG. All three virtual apple trees encoded in RGG graph are loaded from three XEGs generated by the ETL processes of *ClientSideInterface* in the direction of MTG to XEG. The virtual apple tree encoded in MTG at the right side of part A is the original result from the MAppleT simulation. The other two virtual apple trees encoded in MTG on the right side of part B and C are converted from the two corresponding virtual apple trees encoded in RGG graph on the left via XEGs (through an ETL process from RGG graph to XEG and an ETL process from XEG to MTG). Figure 6.6 shows partially the topology of the RGG graph converted from the XEG encoding the small apple tree (the topology embodying the decomposition scheme of 'leaf' metamer nodes M1 and M2 can be directly viewed). Such an experiment geometrically validates the ETL processes at both client and server sides.

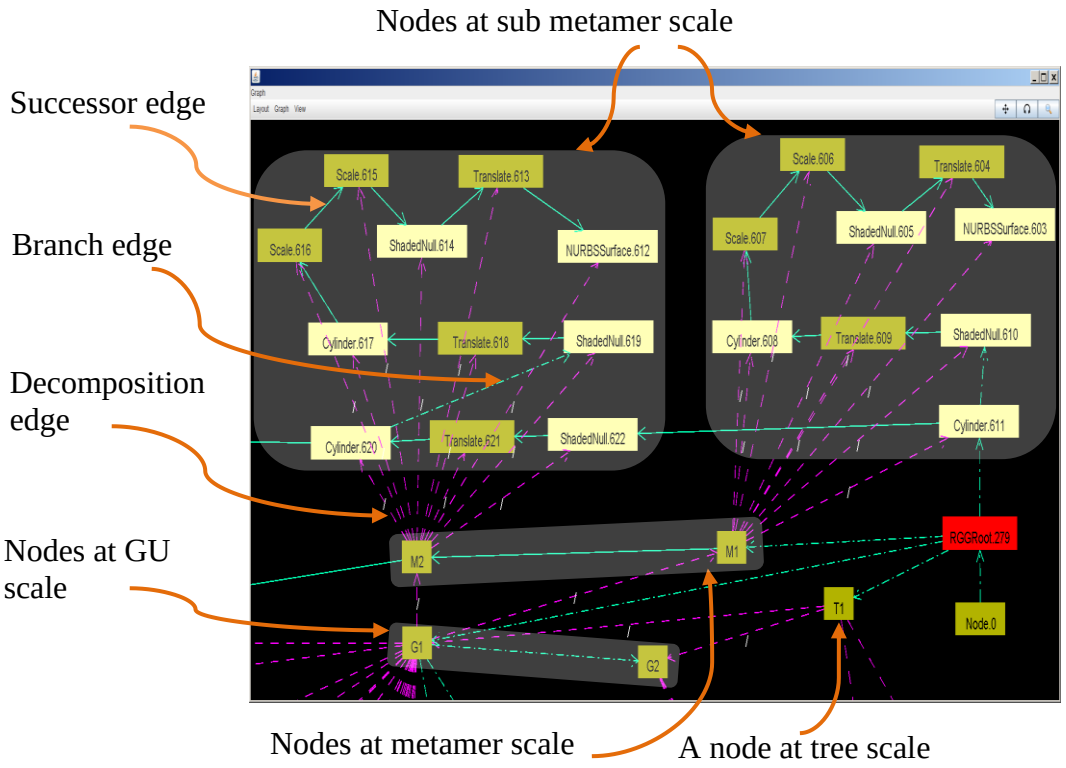


Figure 6.6 The topology of the RGG graph converted from an XEG encoding a small apple tree from MAppleT shown in 2D on GroIMP

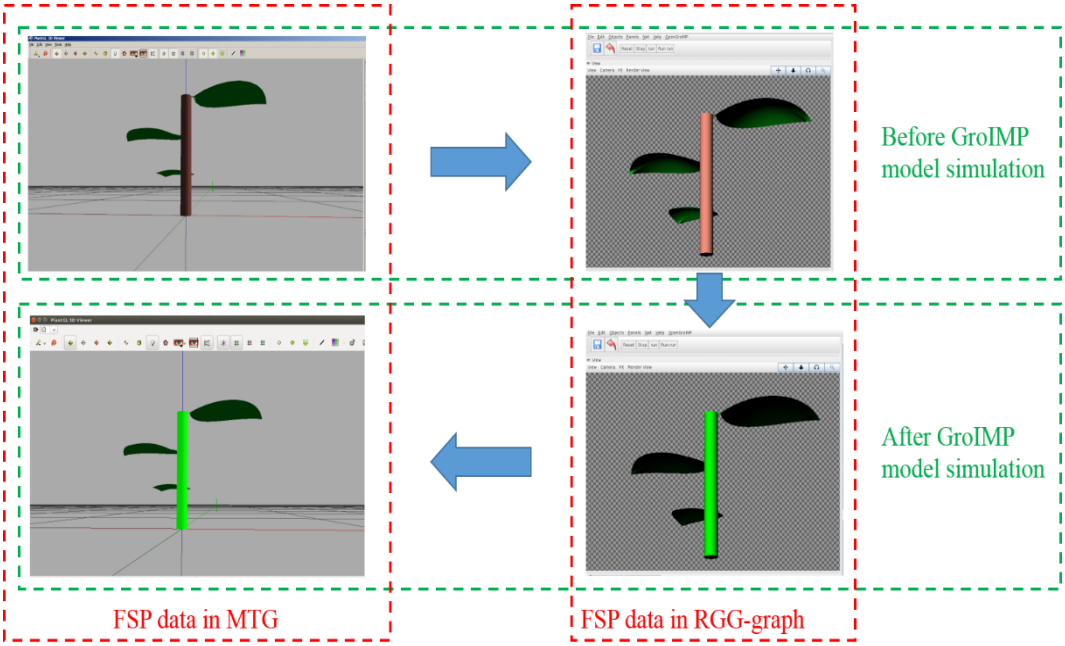


Figure 6.7 Experiment to test the interface by a GroIMP color-changing model. The arrow points to show the flow of data between different data models and FSPM.

We tried a third experiment to test the interface by applying a simulation of a simple GroIMP model through ETL processes. We used a GroIMP model with a single production rule to change the color of plant modules representing internodes, then loaded the modified FSP data to XEG and sent them back to *ClientSideInterface*. Thus the XEG appears to correctly present the plant structure with modified color. Figure 6.7 shows that the plant structure is correctly represented in different MTGs and RGG graphs with modified color. This experiment approved that the FSP data passed through ETL processes by the interface can be further processed by a GroIMP model, and the processed results with updated data can be correctly re-encoded in MTG for MAppleT's further simulation. One remark is that the nodes in a RGG graph are initially of a graphic type without functional properties, which is exactly the case for XEG as it is converted from MTG. It is necessary to allow the simulated functional properties

to be stored in the RGG graph imported from XEG. We first came up with the approach to replace an original node by a module type that extends the type of the original node. This approach complies with the GroIMP method to define a plant module with functional properties but it brings some inconveniencies. In fact, after the GroIMP model simulation, the result needs to be converted back to XEG. At its finest scale, the nodes with functional properties are of extended module types. The interface converts the nodes to objects of PlantGL types to form an object of PlantGL *Scene* type. That means the properties need to be moved and added up to nodes at metamer scale. Therefore, we think it is appropriate to directly sum up the values of all properties of nodes at the finest scale that are decomposed from a metamer node to be summed to the corresponding properties of the metamer node. Such a process avoids the type extension and is actually a property upscaling. It is necessary to have such property upscaling for the retroactive simulation because the simulated result needs to be converted back to MTG which has its properties carried not by graphic objects but by nodes at metamer or coarser scales. Figure 6.8 shows an example of property upscaling that sums up the property 'interceptedLightAmount' from four different nodes (G1, G2, G3 and G4) at growth unit scale to a node (T1) at tree scale. Notice that this particular method was chosen because it meets our specific needs. There are various other methods for property upscaling besides this method.

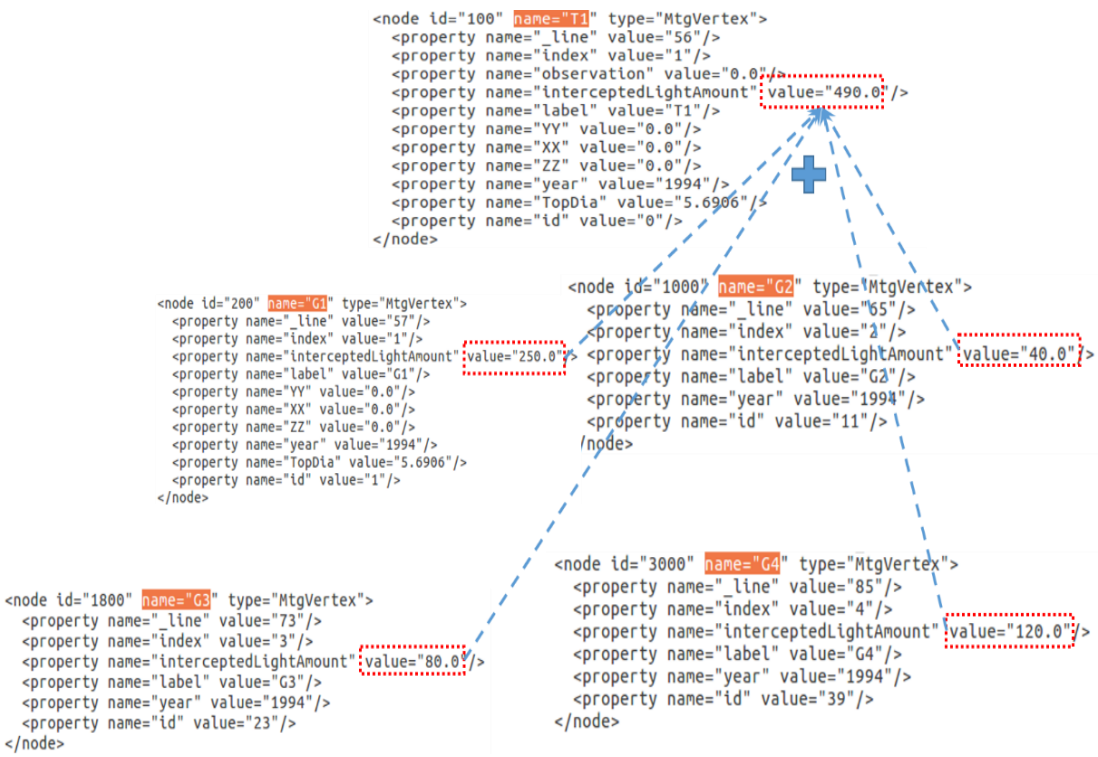


Figure 6.8 An example of property upscaling.

After the experiments, we can now perform a simulation of the integrated FSPM by using the interface. Through the interface, MAppleT and the GroIMP transport model were supposed to be integrated as one model that was supposed to be able to simulate apple tree growth by considering the water and sugar conditions. However, we have changed the simulation plan for internal reasons. The new plan was to integrate a GroIMP light model with MAppleT. MAppleT is a stochastic model that does not take any functional conditions except gravity to compute the structure of apple trees. An intermediate OpenAlea FSPM that takes light conditions to compute the photosynthesis is planned to do a pre simulation before that done by MAppleT. MAppleT then takes over the tree structure with properties of assimilate content from photosynthesis to play with the size of each plant module. In such a way, the project can be validated and the concept of the integration of different FSPMs can be approved as well.

A *Module* that extends the ray tracer based GroIMP type *SpotLight* is used in the light model as the type of light source. We set up a particular position for the light, and varied the light energy (i.e. the power in Watt) and the number of rays. We want to compare the MAppleT simulated structure by different light energy (same numbers of rays), and by different numbers of rays (same light energy). The light model computes the amount of light being absorbed by the plant modules and we assumed that only green plant modules can intercept light.

So far, we are still working at the simulation of the integrated FSPM through the interface in cooperation with our French partners.

6.3 The enhancements of GroIMP and the interface

During the application of the interface, we have found that the RGG three-part graph lacks possibilities for query in a multiscale manner for the usage in XL imperative code blocks. We thus developed some query commands to bridge the difference between the true graph topology and the multiscale topology to allow the graph to be used as a 'real' multiscale graph.

- *findParent (Node n)*: find the parent nodes of the parameter node
- *findChildren (Node n)*: find the children nodes of the parameter node
- *findComplex (Node n)*: find the coarse node that decomposes into the parameter node
- *findComponents (Node n)*: find the nodes into which the parameter node decomposes

- *findFinestComponents (Node n)*: find the nodes at finest scale that are accessible by paths of decomposition edges from the parameter node.

- *findAllNodesAtSameScale (Node n):* find all nodes at the scale that the parameter node locates

- *getScaleNumber (Node n)*: get the index of the scale where the parameter node locates. Such index is defined to identify the topological position in an RGG graph with linearly ordered scales. In other words, such indexing concept only applies to a specific variant of the RGG graph.

- *getRootsAtScale (Int scaleNumber)*: get the root node at a scale specified by the parameter scaleNumber

- *getNodesAtScale (Int scaleNumber):* get all nodes at a scale specified by the parameter scaleNumber

- *getMaxScaleNumber ():* get the maximum number of scales for the current RGG graph.

- *getGraphScaleNumber ():* get the number of scales for the current RGG graph. (normally a universal root represents the scale of the whole graph, with number 0, consequently *graphScaleNumber =getMaxScaleNumber +1*)

- *getGraphRoot():* get the root node of the current RGG graph

Besides, the ETL components at the server side effectively make the XEG become a possible data format ideally suited for FSPMs based on GroIMP to preserve its simulation results and to restart simulation using the preserved results. Thus, we have developed software modules to allow the manual import and export of XEG through graphic user interface (GUI, c.f. Figure 6.9). This calls the ETL processes just like the component ‘Message’ does.

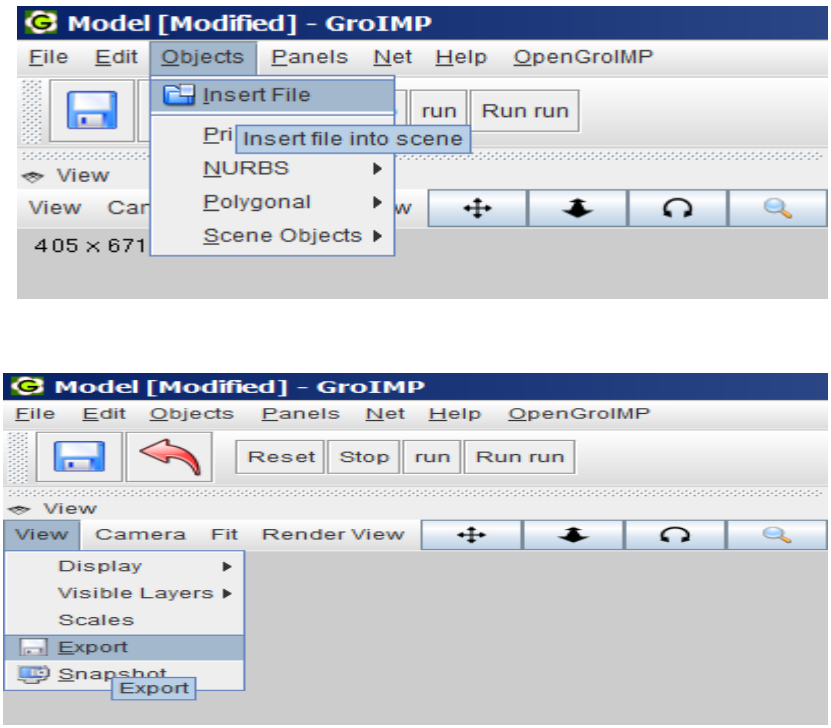


Figure 6.9 GUI for manual import (top) and export (bottom) of XEG

To facilitate the usage of the *ClientSideInterface* (with a given name ‘groalea’ to be integrated as a part of the OpenAlea package), we developed graphical components to allow different FSPMs to be integrated by the usual way of OpenAlea, i.e. to form a workflow by drag/drop of the components and connecting them with edges. Some workflows for the integration case of our projects are also provided as examples (c.f. Figure 6.10).

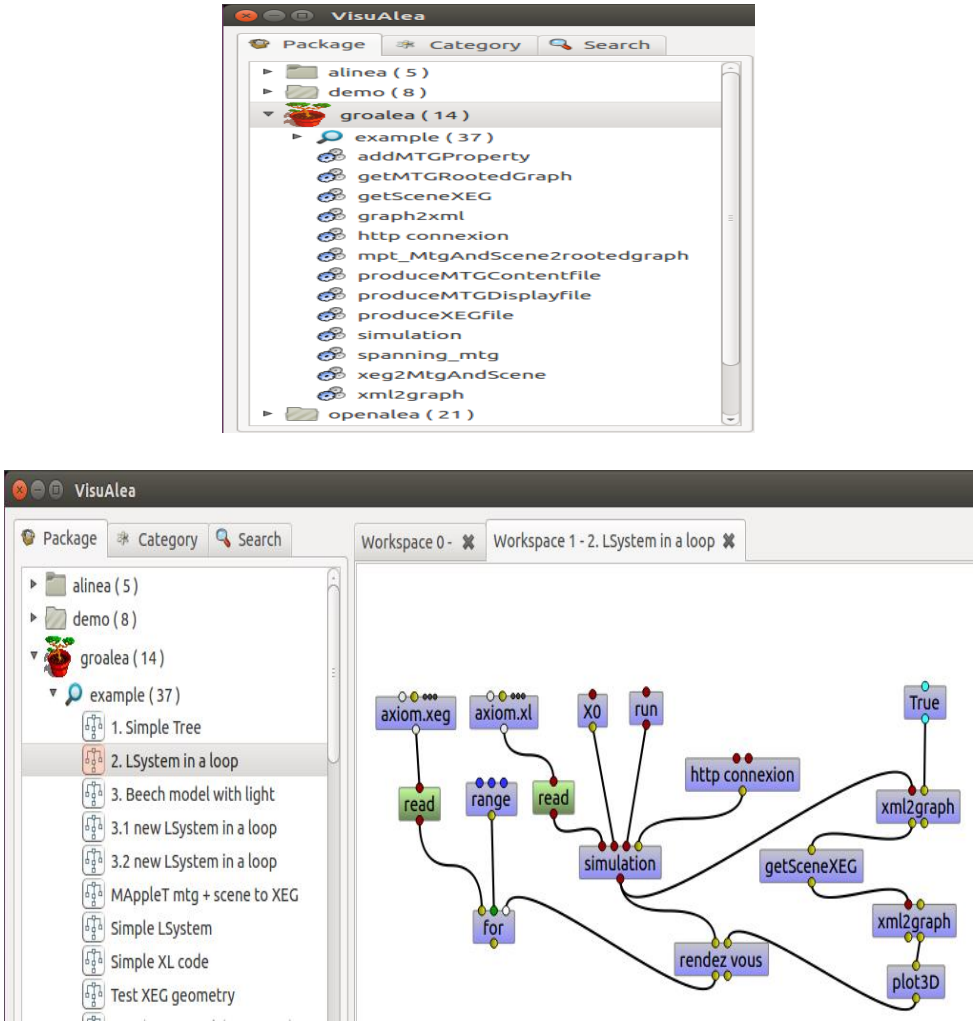


Figure 6.10 Graphical components (top) of groalea and an example of visual workflow (bottom) to run a FSPM at server side constructed using the graphic components.

To facilitate the usage of the *ServerSideInterface*, a GUI module (c.f. Figure 6.11) to allow the server to be launched has also been provided. After it is launched, the client can start the integrative simulation for both retroaction and non-retroaction.

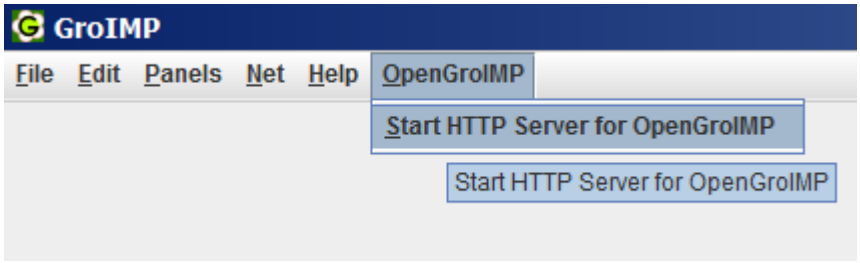


Figure 6.11 GUI components on GroIMP to launch the integrative server

6.4 Discussion and conclusions

The designed middleware technology - FSPM integrative RPC protocol, the component model, and the C/S-ETL based architecture provide a comprehensive and generic technical framework for the integration of different FSPMs. Its effectiveness has been approved through the developed interface for our specific project. This enables the integration of MAppleT with a FSPM based on GroIMP and further integration of FSPMs based on the two platforms. Hence, we conclude that our design and developments fully fulfill the objective of the FSPM Apple project and the PhD tasks. On the other hand, we have also witnessed some collision for the concept of the integration of different FSPMs. In our project, the objective is to have an integrated model that simulates growth of an apple tree considering the water and sugar transport. However, MAppleT itself is a stochastic model that does not take any functional aspects to compute the growth of apple trees. So to allow the integration, MAppleT needs to 'modify' its core production rules, from stochastically based to biological function based. In the application, even another model computing photosynthesis is used to bridge the gap between apple tree growth and intercepted light. Although the component based structure coming with the integration enables high flexibility and some of the developed components are reusable for further integration, such a situation of collision indicates the concept

of integration of different FSPMs might be valid in general, but its adaptability to specific projects needs to be verified beforehand. Moreover, estimations will have to be made to compare the cost of the integration of different FSPMs with the alternative of directly enriching the existing FSPMs by adding modules.

Beside of our project, some other project attempting to integrate different FSPMs have been carried out as well, including the integration of FSPMs based on GroIMP and PyGMAlion (Plant Growth Modeling Analysis and Identification) [158], the integration of FSPMs based on OpenAlea and a specific FSPM called RATP (Radiation Absorption, Transpiration and Photosynthesis) [159], and the integration of FSPMs based on OpenAlea and the Lignum model [160].

PyGMAlion is a platform that provides assistance to the development of FSPMs, mainly through model comparison and selection [161], parameter estimation [162], sensitivity analysis [163], uncertainty analysis [164]. An interface [165] was established to allow the communication between FSPMs based on GroIMP and PyGMAlion for sensitivity analysis. The communication is based on an exchange of data that is managed by a CSV (Comma-Separated Values) based simple data model, where FSPMs based on GroIMP provide simulation inputs and outputs for the sensitivity analysis by PyGMAlion. From the point of view of application, the integration through the interface is not exactly an integrative simulation of plant growth like the integration through our interface, but about the analysis of the relationships between specific inputs and outputs of a FSPM. The exchanged data is not functional and structural plant data, i.e. an instance of an FSP graph, but values of some specific numerical properties. Thus, no structural (topology and geometry) alignment is involved. From the point of view of data/information interoperability, the integration is different from ours. On the other hand, the integration involves process interoperation with fixed syntax, including the syntax of run.sh and commands to run a sensitivity analysis algorithm. The user of the interface determines the semantics, i.e. the meaning of a specific interoperation

between an FSPM model and a sensitivity analysis algorithm. Thus, from the point of view of process interoperability, this integration is similar to ours. Overall, the integration can be regarded as the non-retroactive case of our case.

RATP is a FSPM for light interception, water consumption, and carbon allocation of a tree using the Beer Lambert law, transpiration and photosynthesis models. It has been enclosed as a component provided to the OpenAlea users for building FSPMs. The model was developed in the end of last century by Fortran 90 language. The tree structure is represented by 3D raster graphics, i.e. an array of 3D voxels that does not capture explicit neighboring relationships (topology) between plant modules (functional units). It was enclosed in OpenAlea through the mechanism of two-way conversion between the 3D array of voxels and MTG. The essence of such mechanism is to provide the data/information interoperability between RATP and the other FSPMs by taking the MTG as a canonical data model. Besides, the mechanism provides Python functions to allow the communication between RATP and other FSPMs. Thus from the point of view of interoperability of both data/information and processes, the integration is similar to ours. However, the canonical data model originated from a specific need with a tree structure with separated topology and geometry, while the communication is based on some specific function available on OpenAlea (in Python), which greatly reduces its adaptability. Thus such mechanism is limited to the usage of enclosing models in the OpenAlea platform and enabling the integration of FSPMs through OpenAlea.

Lignum is an FSPM developed in the late 1990s and a XML based data format was included in 2006. The model includes a standard overcast sky based light model to allow the modeling of the interaction between plant structure (growth) and functions (light interception and photosynthesis). An interface [166, 167] has been created to allow the model to be executed under OpenAlea so that the light model can be compared with the light models available on OpenAlea (e.g. RATP). Through the interface, the MTG and Lignum XML based data models are

interoperable in both directions. This integration is a simplified version of our integration, which enables the data/information interoperability by direct conversion without canonical data model as intermediate. Thus, the interface is dedicated to this specific integration and cannot be reused by other integration projects. Moreover, the integration does not involve the interoperability of processes, thus the interface is indeed only for the comparison of the simulation results of different models, not for a cooperative simulation of different models.

In addition to the integration of FSPMs, OpenAlea has also been enriched by integrations using wrapper tools such as Boost.Python [168], SWIG (Simplified Wrapper and Interface Generator) [169] to integrate C/C++ based libraries to Python based platforms, F2PY [170] to integrate Fortran based libraries to Python based platforms. A typical example is PlantGL that was originally developed in C++ and has been integrated [55] to OpenAlea as its basis of geometrical modeling of plants using the wrapper tool Boost.Python. This kind of integration is the integration of modeling tools, not of the FSPMs themselves. Unlike the integration of FSPMs, where the FSPMs are software with comparable functions (FSP data processing) with comparable technologies applied, the tools have incomparable functions with incomparable technologies applied (graphics library for 3D modeling, MTG for multiscale topology modeling). Thus, we have here are actually general software integrations, or more precisely a software composition. For this, a general canonical data model is not possible and the integration is limited by the specific technologies, e.g., wrappers, with an interoperability that is limited to the technical level.

Through the comparison between the other integrations and ours, we can conclude that our integration indeed provides a generic solution at both methodological and technical levels for the integration of a specific type of software, i.e. different FSPMs. To illustrate the integration capabilities of the designed techniques and the implemented interface, we compare integration

solutions based on our techniques with an example hosted on a single platform, an FSPM called MuSCA.

MuSCA is a multiscale FSPM to compute carbon allocation at different user defined spatial scales, allowing the comparison of results and estimation of the impact of the scale setting [171] . It is a modular model that takes use of some components on and through OpenAlea, including MTG (with PlantGL) as its data model and RATP for light interception. Besides, it simulates biomass accumulation using a carbon flow model that represents the flow as a function of source and sink inversely related to distance and resistance (friction) in-between. The RATP light interception model computes the absorbed light amount using 3D voxels/raster graphics which is less precise than the vector graphics based FSPM such as GroIMP ray tracer based models, while the carbon flow model considers only the distance and friction which is less precise than FSPMs based on Munch flow [136, 172]. Consequently, MuSCA can only simulate a less precise biomass accumulation and provides a rough assessment for different scale settings. If our interface is applied, the intercepted light amount and biomass accumulation can be more precisely computed by the integrated FSPMs available on GroIMP.

On the other hand, the designed technologies and the implemented interface have also some limitations. For the designed technologies, the major one is the lack of measures to allow simultaneous simulations. In other words, the current design and implementation allows only a sequential integrative simulation of different FSPMs. Besides, we designed the component model with an architecture that has the component *ConfManager* at client side to allow the plant scientists to input the configuration setup for the integration based on biological knowledge, without providing any standard to define the inputs. Both the two limitations are caused by lack of biological background and high diversity of different FSPMs. We think some sort of standardized input format to allow the coordination of simulation of different FSPMs can be a realizable target. For the implemented interface, it

certainly has the limitations of the used technologies. Another major limitation for the interface is that the integrative protocol is not standardized yet, thus it is limited to the usage of OpenAlea and GroIMP based FSPMs, and it is still a middleware with regard to FSPMs. This means that the FSPMs based on platforms that are different from the two need to have a library to support XEG processing before they can be integrated using the protocol. Such kind of library in different programming environments can be provided as an alternative way of protocol standardization. Finally, our interface has the potential to enable a two-way integration of FSPMs based on GroIMP and OpenAlea. To turn such potential into reality, the client at GroIMP side, the server at OpenAlea side, and the extension of the implementation for the map for division of XEG encoding single scale FSP data to allow both geometrical and non-geometrical XEG to be generated is needed.

Chapter 7

APPENDICES

7.1 The technical documents of the interface for the integration of target FSPMs

7.1.1 The specification of XEG

The data model XEG is specified mainly by an XML schema [173]:

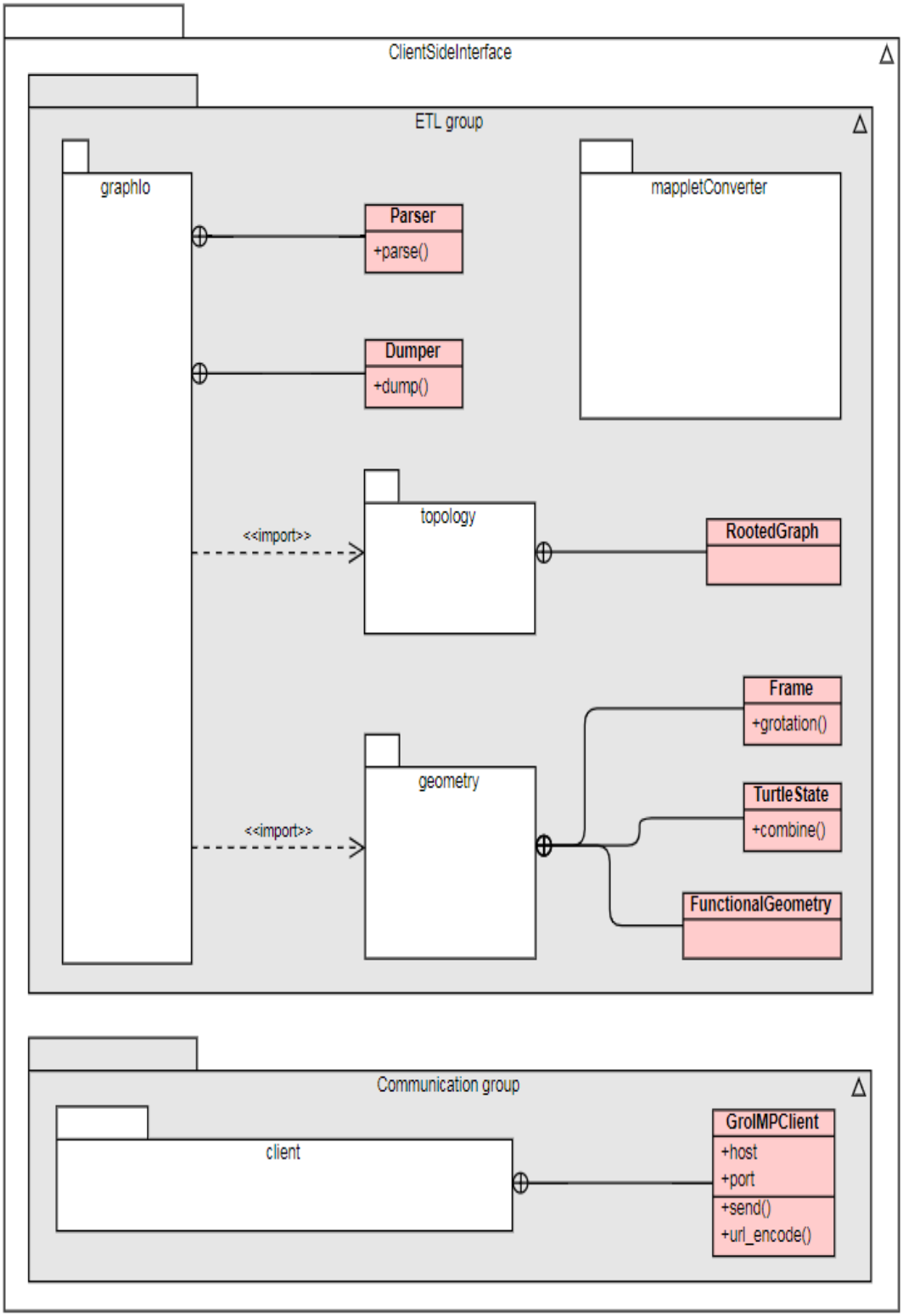
```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="graph" type="Graph" />
      <xs:complexType name="Graph">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:element name="type" type="Type" minOccurs="0"
maxOccurs="unbounded" />
           <xs:element name="root" type="Root" minOccurs="0" maxOccurs="1" />
                  <xs:element name="node" type="Node" minOccurs="0"
maxOccurs="unbounded" />
                  <xs:element name="edge"   type="Edge" minOccurs="0"
maxOccurs="unbounded" />
            </xs:sequence>
      </xs:complexType>
```

```
<xs:complexType name="Edge">
	<xs:attribute name="id" type="id_type" use="optional" />
	<xs:attribute name="src_id" type="id_type" use="required" />
	<xs:attribute name="dest_id" type="id_type" use="required" />
	<xs:attribute name="type" type="xs:string" use="required" />
</xs:complexType>

<xs:simpleType name="float_type">
	<xs:restriction base="xs:float" />
</xs:simpleType>

<xs:simpleType name="int_type">
	<xs:restriction base="xs:int"/>
</xs:simpleType>

<xs:simpleType name="list_of_float_type">
	<xs:list itemType="float_type" />
</xs:simpleType>

<xs:simpleType name="list_of_int_type">
	<xs:list itemType="int_type" />
</xs:simpleType>

<xs:simpleType name="float4x4_type">
	<xs:restriction base="list_of_float_type">
		<xs:minLength value="16" />
		<xs:maxLength value="16" />
	</xs:restriction>
</xs:simpleType>

<xs:simpleType name="matrix_type">
	<xs:annotation>
		<xs:documentation>
			Matrix transformations embody mathematical changes to
			points within a coordinate systems or the coordinate
			system itself. The matrix element contains a 4-by-4
			matrix of floating-point values.
		</xs:documentation>
	</xs:annotation>
	<xs:restriction base="float4x4_type" />
</xs:simpleType>

<xs:simpleType name="float3x1_type">
	<xs:restriction base="list_of_float_type">
		<xs:minLength value="3"/>
		<xs:maxLength value="3"/>
	</xs:restriction>
</xs:simpleType>
```

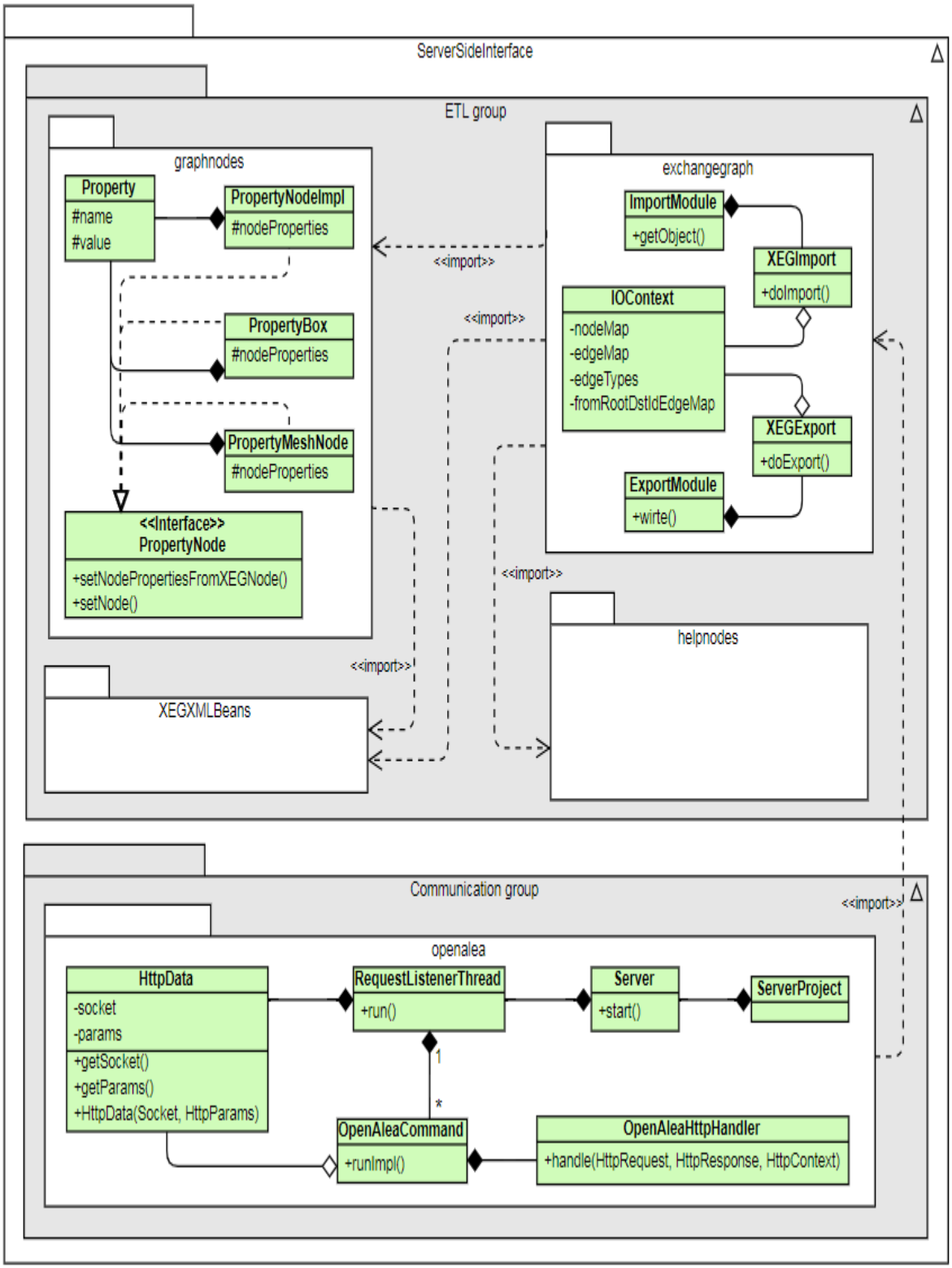
```
<xs:simpleType name="float4x1_type">
	<xs:restriction base="list_of_float_type">
		<xs:minLength value="4"/>
		<xs:maxLength value="4"/>
	</xs:restriction>
</xs:simpleType>

<xs:simpleType name="rgb_type">
	<xs:restriction base="float3x1_type"/>
</xs:simpleType>

<xs:simpleType name="rgba_type">
	<xs:restriction base="float4x1_type"/>
</xs:simpleType>

<xs:simpleType name="list_of_float">
	<xs:restriction base="list_of_float_type"/>
</xs:simpleType>

<xs:simpleType name="list_of_int">
	<xs:restriction base="list_of_int_type"/>
</xs:simpleType>

<xs:simpleType name="id_type">
	<xs:restriction base="xs:long" />
</xs:simpleType>

<xs:complexType name="Node">
	<xs:sequence maxOccurs="unbounded" minOccurs="0">
		<xs:element name="property" type="Property" />
	</xs:sequence>
	<xs:attribute name="id" type="id_type" use="required" />
	<xs:attribute name="type" type="xs:string" use="optional" />
	<xs:attribute name="name" type="xs:string" use="optional" />
</xs:complexType>

<xs:complexType name="Property">
	<xs:choice minOccurs="0" maxOccurs="1">
		<xs:element name="matrix" type="matrix_type" />
		<xs:element name="rgb" type="rgb_type" />
		<xs:element name="rgba" type="rgba_type" />
		<xs:element name="list_of_int" type="list_of_int" />
		<xs:element name="list_of_float" type="list_of_float" />
	</xs:choice>
	<xs:attribute name="name" type="xs:string" use="required" />
	<xs:attribute name="value" type="xs:string" use="optional" />
	<xs:attribute name="type" type="xs:string" use="optional" />
</xs:complexType>
```

```
	<xs:complexType name="Type">
		<xs:sequence minOccurs="1" maxOccurs="1">
			<xs:element name="extends" type="ExtendsType"
minOccurs="1" maxOccurs="1" />
			<xs:sequence minOccurs="0" maxOccurs="unbounded">
				<xs:element name="implements"
type="ImplementsType" />
			</xs:sequence>
			<xs:sequence maxOccurs="unbounded" minOccurs="0">
				<xs:element name="property" type="Property" />
			</xs:sequence>
		</xs:sequence>
		<xs:attribute name="name" type="xs:string" />
	</xs:complexType>

	<xs:complexType name="ImplementsType">
		<xs:attribute type="xs:string" name="name" />
	</xs:complexType>

	<xs:complexType name="ExtendsType">
		<xs:attribute type="xs:string" name="name" />
	</xs:complexType>

	<xs:complexType name="Root">
	  <xs:annotation>
		<xs:documentation>
			Root is an extra node and does NOT refer to an
			existing node in the node array. It is used with its ID
			to model the edges. This node is mapped in GroIMP as an
			object of class Node.
		</xs:documentation>
	  </xs:annotation>
	  <xs:attribute name="root_id" type="id_type" use="required">
	  </xs:attribute>
	</xs:complexType>
</xs:schema>
```

7.1.2 The package diagram of the *ClientSideInterface*



7.1.3 The package diagram of the *ServerSideInterface*



7.2 The user manual of the interface

7.2.1 The installation of the interface

The interface has been developed on top of GroIMP and OpenAlea, as a middleware between platforms and FSPMs. It consists of two parts, the *ClientSideInterface* on top of OpenAlea and the *ServerSideInterface* on top of GroIMP. Each part is supposed to be included in the new version of the corresponding platform. Before that, the two parts of the interface have to be installed separately, in a developer mode, under Linux (Ubuntu is recommended). To install the two parts of the interface in developer mode, the pre-condition is to check out the code of each corresponding platforms.

For *ClientSideInterface* (named 'groalea'), the source code of the most recent version of the OpenAlea platform needs to be checked out from its official repository. However, the source code of the most recent version did not work correctly during the project. We have used the source code with three packages (Openalea-1.2.0.tar.gz, VPlants-1.2.0.tar.gz, Alinea_1_0.tar.gz) of an earlier version (release 0.9) from this webpage:

(http://openalea.gforge.inria.fr/wiki/doku.php?id=download:source_distribution).

Once the three packages are retrieved to the local operation system, they can be installed by following the instructions for Ubuntu 12.10 under the section Compilation from sources at this webpage:

(http://openalea.gforge.inria.fr/wiki/doku.php?id=documentation:user:ubuntu#dependencies).

When the three packages of the OpenAlea platform release 0.9 have been installed correctly, the interface can be installed by following the steps below:

1. Clone the remote repository to local using the command

   ```
   git clone https://github.com/longmanplus/groalea.git
   ```

2. Go to the directory of the local package where the setup.py is located.

3. Install groalea using the command

   ```
   python setup.py install
   ```

   Or, to contribute on the interface, use the command

   ```
   python setup.py develop
   ```

General instructions for the working on a github project can be found here:

http://virtualplants.github.io/contribute/devel/workflow-github.html#workflow-github

For the *ServerSideInterface*, the source code of the most recent version of the GroIMP platform needs to be checked out from its official repository, on SourceForge. The IDE Eclipse is recommended. The source code of GroIMP can be found here: https://sourceforge.net/p/groimp/code/HEAD/tree/ . To install the platform, all packages except ExchangeGraph, OpenAlea, Graph, RGG need to be checked out from the trunk using Eclipse (c.f. Figure 7.1). (Note the interface includes the first two packages. The package *Graph* has the *dupnew* method added to /Graph/src/de/grogra/graph/impl/Node.java for duplication of nodes and properties of a node, the package *RGG* has new query methods for multiscale RGG graphs. They enable geometrical upscaling.)

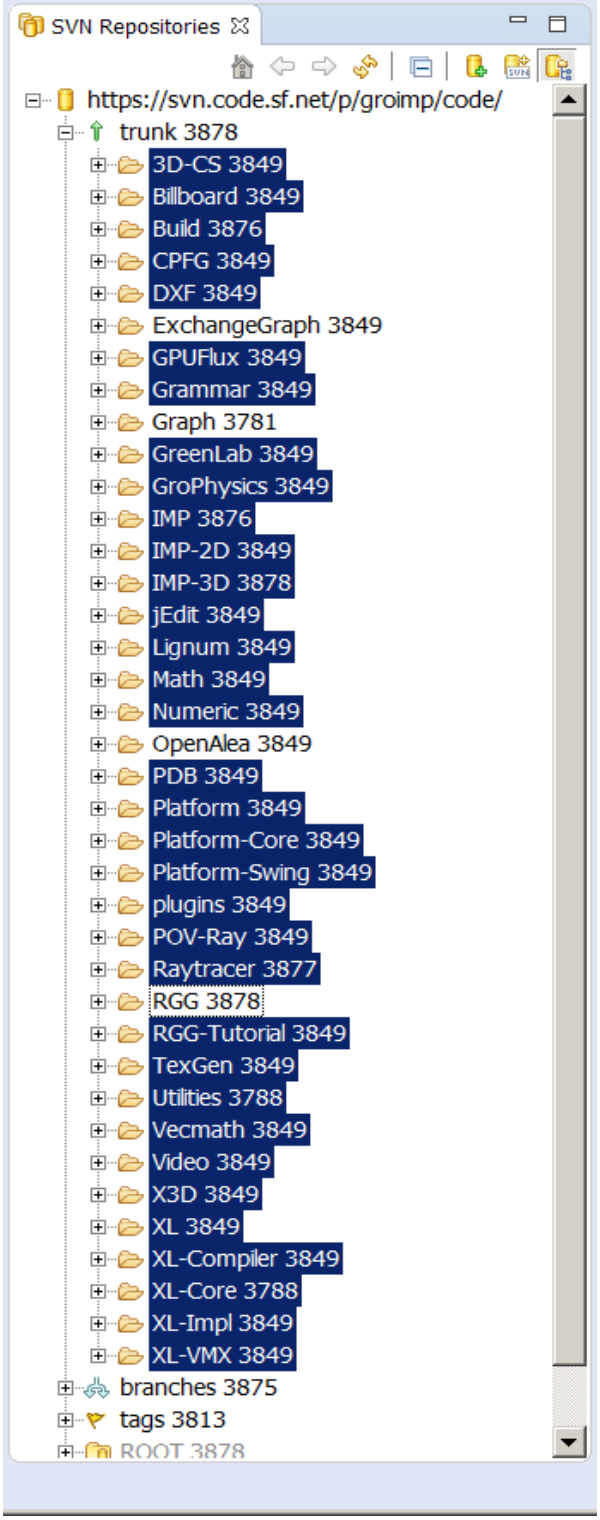


Figure 7.1 The packages/files to be checked out from the trunk of the GroIMP SVN repository

Then, the source code of the interface needs to be checked out from the branch named FSPM Apple (c.f. Figure 7.2) by firstly selecting the files, then clicking on the Check Out item on the right-click menu.

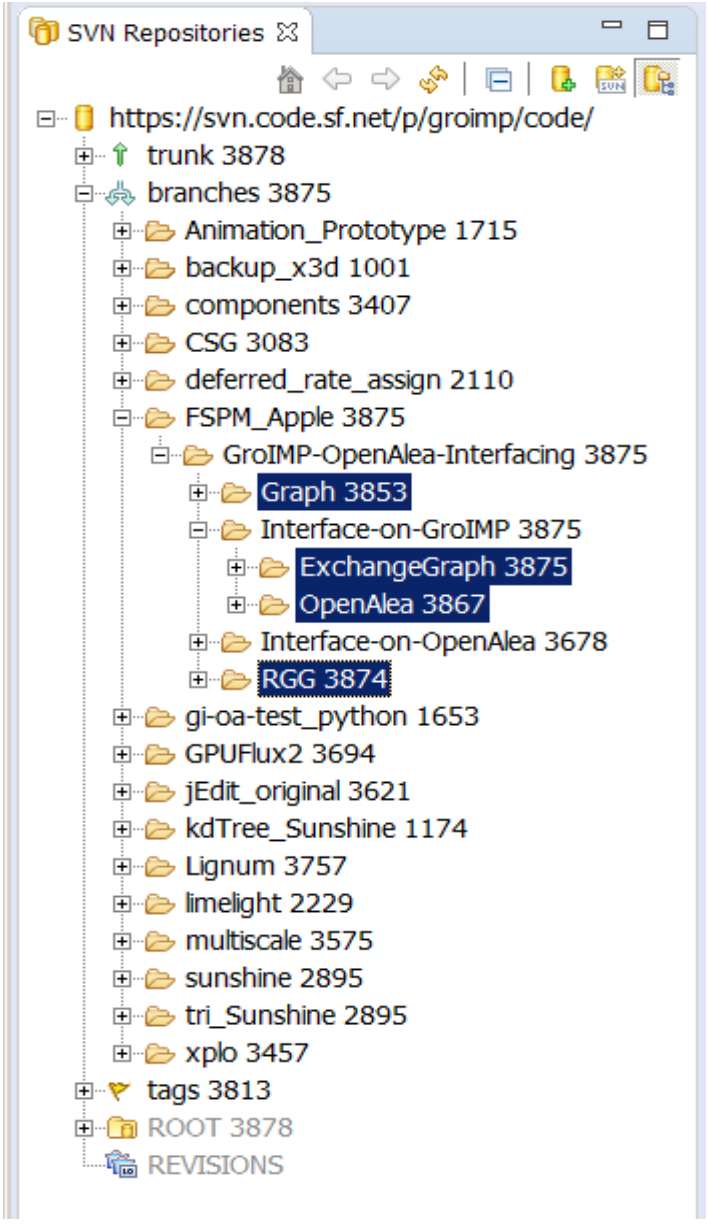


Figure 7.2 The packages to be checked out from the FSPM Apple branch of the GroIMP SVN repository

7.2.2 The usage of the interface

There are three scenarios of using the interface: calling FSPMs based on GroIMP from OpenAlea, integration of the FSPMs, saving simulation results to XEG. These scenarios involve the usage of groalea under OpenAlea except the last one, which mainly refers to the scenario for GroIMP (OpenAlea has already the possibility to save the simulation result in files, i.e. MTG file/.mtg, Scene file/.bgeom).

The usage of groalea under OpenAlea starts by launching OpenAlea by the command visualea. and launching GroIMP by the following instructions.

Click the Run button on the Eclipse main panel, and click Run Configurations… on the appeared drop-down menu. Then right click the item Java Application on the appeared Run Configurations panel, and click the item New on the appeared menu to create a new configuration.

On the appeared panel on the right side, the configuration just created needs to be adjusted with a given name as you like, under the tab Main, the Project must be Platform-Core, Main class must be de.grogra.pf.boot.Main. Under the tab Arguments, the Program arguments must be --project-tree (c.f. Figure 7.3).

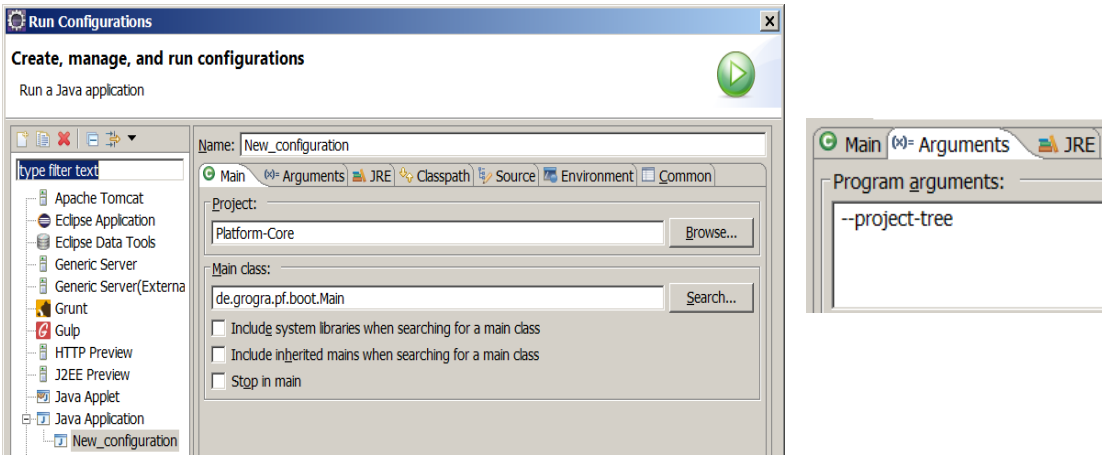


Figure 7.3 Adjustment for the created configuration under Eclipse.

Then the GroIMP platform can be launched by clicking Run button to run the created configuration. On the appeared GroIMP main panel, click OpenGroIMP, then click the item Start HTTP Server for OpenGroIMP (c.f. Figure 6.11), a dialogue box Start OpenGroIMP Server will appear to allow the input of a port number (with pre filled number '58070') . When the port number is provided, the server can be launched by clicking the button OK in the dialogue box.

To call FSPMs based on GroIMP from OpenAlea, continue with the following steps:

1. Construct a workflow similar to LSystem in a loop, where axiom.xeg is an input XEG file, axiom.xl is an input model source code file, 'run' is the name of the 'main' method of the input model. Double click the range box, the range dialogue box with three modifiable value items appears. Those are start, step, and end values of a 'for loop' (c.f. Figure 6.10). The member of message body 'time' is set using the loop values. (Note that we set the default value of the server host and port to 'localhost', i.e. 127.0.0.1, and '58070'). The host can be modified by connecting a string box with given IP value (similar the 'run' box) to the red point on the right side of the box http connection. The default port number is not supposed to be modified, unless a different port number has been agreed upon, and the GroIMP side has launched the OpenGroIMP server with the agreed port number.

2. At the top of the panel Visualea, click on WorkSpace, then click the item Run to run the simulation of the constructed workflow in the current workspace.

To run the integrative simulation of the FSPMs, the pre-condition is to have the .mtg and .bgeom file pairs, or the objects of MTG and Scene types available. The latter case needs to have MAppleT available as a visual component in OpenAlea to

construct a workflow. However, for internal reasons, we currently do not have it available. Thus, the current possibility is to run the integrative simulation manually under the former pre-condition.

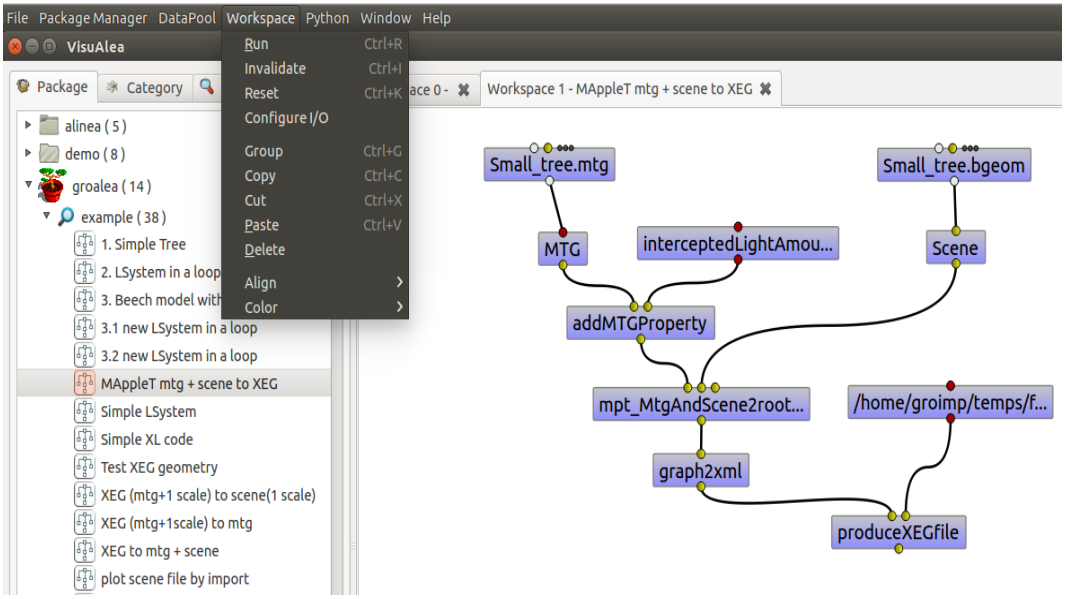


Figure 7.4 The way to run the example workflow MAppleT mtg + scene to XEG

With the available .mtg and .bgeom file pairs, a workflow similar to the example workflow MAppleT mtg + scene to XEG (c.f. Figure 7.4) needs to be created. The box addMTGProperty adds a property with initial value zero to every MTG node except the root with a given name. The box produceXEGfile generates an XEG file with a given full name. Conversely, a workflow similar to the example workflow XEG to mtg + scene (c.f. Figure 7.5) needs to be created. Through the box xeg2MtgAndScene, an XEG file can be converted to an object of MTG type and an object of Scene type. The results can be further processed by MAppleT, or explored (e.g. explored graphically by box plot3D).

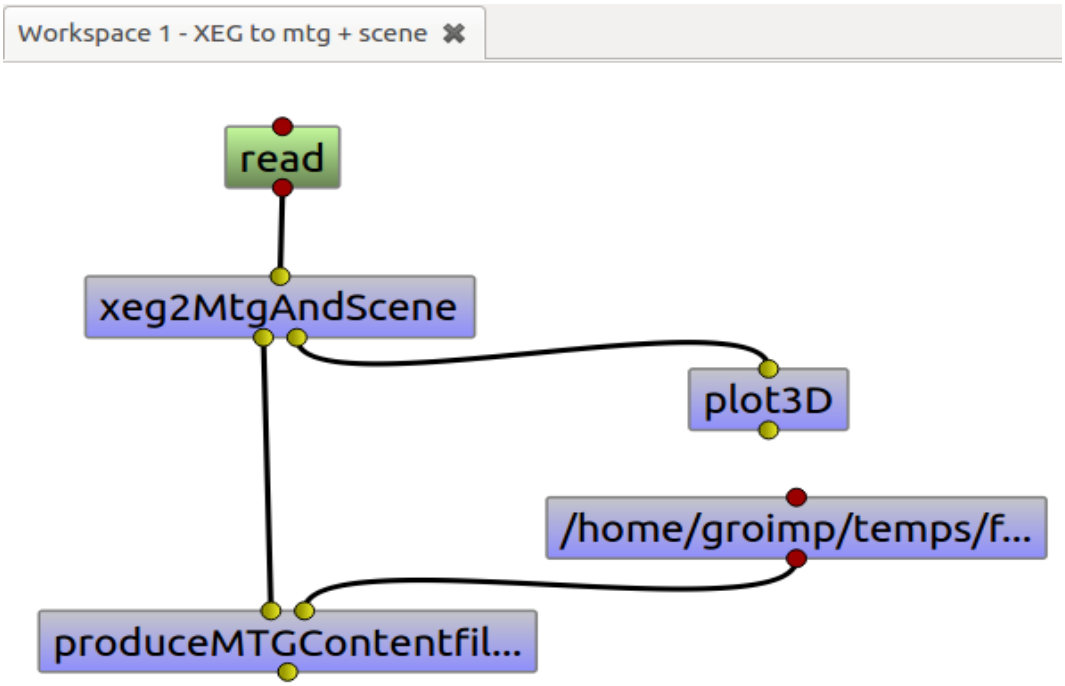


Figure 7.5 The example workflow XEG to mtg + scene

To allow the generated XEG to be processed by a GroIMP model, it needs to be imported to the workbench where the model is compiled. Click on Objects on the main panel of GroIMP, and then click on the item Insert File on the appeared menu. Then, an XEG can be chosen for importing through the appeared Open File dialogue box, just click Open, the XEG file will be imported. Then, the GroIMP model can be run to modify the imported graph. At the end, the graph needs to be exported to an XEG by clicking on View on the View panel, then click on item Export on the appeared menu. Then an XEG can be exported to a chosen directory with a given name through the appeared Export dialogue box. The steps to save simulation results to XEG are essentially the steps to export an XEG file (c.f. Figure 6.9).

7.3 The source code for the experiments of geometrical upscaling

```
import de.grogra.ext.exchangegraph.graphnodes.*;

scaleclass ScaleTree;
scaleclass ScaleGU;
scaleclass ScaleMetamer;
scaleclass ScaleOrgan;

module PropertyMeshNodeN extends PropertyMeshNode;
module PropertyMeshNodeGU extends PropertyMeshNode;
module PropertyMeshNodeT extends PropertyMeshNode;

public void geoUpScale2M ()
[
      n:PropertyNodeImpl,
      (!empty((*n /> ShadedNull*))
      && empty((*n /> />ShadedNull*)))
      ==> m:PropertyMeshNodeN
      {//m.dupnew(getMesh(getComponentBasedConvexhull(n, false)), false, null);
            m.dupnew(getMesh(getComponentBasedABBHull(n, true)), false, null);
            m.setNodeProperties(n.getNodeProperties());
            m.setShader(RGBAShader.BLUE);};
]

public void geoUpScale2GU ()
[
      n:PropertyNodeImpl,
      (!empty((*n /> PropertyMeshNodeN*))
      && empty((*n /> /> />ShadedNull*)))
      ==> m:PropertyMeshNodeGU
      {//m.dupnew(getMesh(getComponentBasedConvexhull(n, false)), false, null);
            m.dupnew(getMesh(getComponentBasedABBHull(n, true)), false, null);
            m.setNodeProperties(n.getNodeProperties());
            m.setShader(RGBAShader.BLUE);};
]
```

```
public void geoUpScale2T ()
[
      n:PropertyNodeImpl,
      (!empty((*n /> PropertyMeshNodeGU*))
      && empty((*n /> /> /> />ShadedNull*)))
      ==> m:PropertyMeshNodeT
      {//m.dupnew(getMesh(getComponentBasedConvexhull(n,false)), false, null);
            m.dupnew(getMesh(getComponentBasedABBHull(n, true)), false, null);
            m.setNodeProperties(n.getNodeProperties());
            m.setShader(RGBAShader.BLUE);};
]

public void addScaleTypeGraph()
[
              RGGRoot ==>> ^ [
                  /> TypeRoot
                  /> {# nt: PropertyMeshNodeT #}
                  /> {# ngu: PropertyMeshNodeGU #}
                  /> {# m: PropertyMeshNodeN #}
                  /> {# tt:Translate ts:Scale tr:ShadedNull
                  sc:Cylinder ss:Sphere sn: NURBSSurface #}]
               [/>SRoot
               /> scaleTree:ScaleTree
               /> scaleGU:ScaleGU
               /> scaleM:ScaleMetamer
               /> scaleOrgan:ScaleOrgan],
               scaleTree +> {# nt #},
               scaleGU +> {# ngu #},
               scaleM +> {# m #},
               scaleOrgan +>{# tt ts tr sc ss sn #};
]
```

Remark: the method *addScaleTypeGraph* has to be executed when all the upscaling methods have been executed. The source code allows geometrical upscaling from sub metamer scale to metamer (M), growth unit (GU) and tree (T) scales using Convexhull or Axis-aligned Bounding Box based on convex hull (ABBHull). To allow the former, it is necessary to use the green line of each upscaling method, and comment out the next line. To visually switch the scales represented by bounding volumes interactively, one needs to click on View on the View panel, and click the item Scales on the appeared menu (c.f. Figure 6.9). In the appeared dialogue box, the view of different bounding volumes of scales can

then be switched by checking or unchecking the corresponding checkboxes (c.f. Figure 6.2, Figure 6.3). This code computes bounding volumes for different scales based on the sub metamer scale, i.e., the finest scale, where the geometrical nodes are located. It is also possible to compute the bounding volumes for a scale iteratively, based on the next-finer scale. This requires the warranty to ensure the finer scale has already a geometry. The computed bounding volumes obtained from the two ways (in both cases, ABBHull and Convex hull) for a given scale are identical because the extreme points are the same and straight lines connect them.

REFERENCES

[1] I. Sommerville, *Software Engineering*. Addison-Wesley Publishing Company, 2011, p. 792.

[2] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990, p. 228.

[3] IDABC, "Draft document as basis for EIF version 2.0," ed: European Communities, 2008, p. 79.

[4] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 6th ed. Pearson, 2012, p. 864.

[5] Oracle Corporation. (2010, 12 Dec 2018). *Message-Oriented Middleware (MOM)*. Available: https://docs.oracle.com/cd/E19316-01/820-6424/aeraq/index.html

[6] M. Jan. (18 Dec 2018). *Pixels and voxels, the long answer*. Available: https://medium.com/retronator-magazine/pixels-and-voxels-the-long-answer-5889ecc18190

[7] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., 2003, p. 480.

[8] Oracle Corporation. (18 Dec 2018). *Introduction to Web Service Technologies*. Available: https://docs.oracle.com/cd/E13224_01/wlw/docs103/guide/webservices/conBasicWebServiceTechnologies.html

[9] H. Kubicek and R. Cimander, "Three dimensions of organizational interoperability," *European Journal of ePractice*, vol. 6, pp. 1-12, 2009.

[10] D. S. Rosenblum. (2001, 10 April 2019). *Interoperability & Middleware* [PDF]. Available: https://www.ics.uci.edu/~taylor/ICS221/slides/interoperability

[11] C. Godin. (2007, 19 Dec 2018). *Manipulating plant architecture as Multiscale Tree Graphs*. Available: http://openalea.gforge.inria.fr/wiki/doku.php?id=documentation:demo:mtg_reconstruction

[12] C. Godin, "Representing and encoding plant architecture: A review," *Ann. For. Sci.*, vol. 57, no. 5, pp. 413-438, 2000.

[13] K. Smoleňová. (2010, 10 April 2019). *Introduction to rule-based modelling with GroIMP* [PDF]. Available: http://www.sccg.sk/~smolenova/elearning/ks_ecp10.pdf

[14] Q. Long, W. Kurth, C. Pradal, V. Migault, and B. Pallas, "An Architecture for the Integration of Different Functional and Structural Plant Models," in *Proceedings of the 7th International Conference on Informatics, Environment, Energy and Applications*, Beijing, China, 2018, pp. 107-113: ACM.

[15] Y. Ong, K. Streit, M. Henke, and W. Kurth, "An approach to multiscale modelling with graph grammars," *Annals of Botany*, vol. 114, no. 4, pp. 813-827, 2014.

[16] P. T. Cox and S. Baoming, "A formal model for component-based software," in *Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments (Cat. No.01TH8587)*, 2001, pp. 304-311.

[17] Q. Long and W. Kurth, "A logical data exchange model for adapting different methods abstracting plant architecture," in *2017 2nd International Conference on Knowledge Engineering and Applications (ICKEA)*, 2017, pp. 39-43: IEEE.

[18] C. Godin and H. Sinoquet, "Functional–structural plant modelling," *New Phytologist*, vol. 166, no. 3, pp. 705-708, 2005.

[19] J. Vos, J. B. Evers, G. H. Buck-Sorlin, A. Bruno, C. Michael, and P. H. De Visser, "Functional-structural plant modelling: A new paradigm in crop science," *Comparative Biochemistry and Physiology a - Molecular & Integrative Physiology*, vol. 153a, no. 2, pp. S223-S223, Jun 2009.
[20] J. Vos, L. Marcelis, and J. Evers, "Functional-structural plant modelling in crop production: adding a dimension," *Wageningen UR Frontis Series*, vol. 22 Functional-Structural Plant Modelling in Crop Production, pp. 1-12, 2007.
[21] IDABC, "European interoperability framework for pan-European e-government services v1.0," ed. Luxembourg: European Communities, 2004, p. 26.
[22] G. Buck-Sorlin, "Functional-Structural Plant Modeling," in *Encyclopedia of Systems Biology*, W. Dubitzky, O. Wolkenhauer, K.-H. Cho, and H. Yokota, Eds. New York, NY: Springer, 2013, pp. 778-781.
[23] R. Sievanen, C. Godin, T. D. DeJong, and E. Nikinmaa, "Functional-structural plant models: a growing paradigm for plant studies," *Annals of Botany*, vol. 114, no. 4, pp. 599-603, Sep 2014.
[24] B. Bayol, P. H. Cournède, J. Sainte-Marie, G. Viaud, F. Chi, W. Kurth, Q. Long, J. Merklein, K. Streit, E. Costes, V. Migault, B. Pallas, G. Buck-Sorlin, M. Poirier-Pocovi, and C. Pradal, "Multiscale functional-structural plant modelling at the example of apple trees: Project description," in *2016 IEEE International Conference on Functional-Structural Plant Growth Modeling, Simulation, Visualization and Applications (FSPMA)*, Qingdao, China, 2016, pp. 1-5: IEEE.
[25] W. Kurth, G. Buck-Sorlin, E. Costes, and P.-H. Cournède, "Multiscale functional-structural plant modelling at the example of apple trees," Unpublished Project Proposal, 2014.
[26] E. Costes, C. Smith, M. Renton, Y. Guédon, P. Prusinkiewicz, and C. Godin, "MAppleT: simulation of apple tree development using mixed stochastic and biomechanical models," *Functional Plant Biology*, vol. 35, no. 10, pp. 936-950, 2008.
[27] T. M. DeJong, D. Da Silva, J. Vos, and A. J. Escobar-Gutierrez, "Using functional-structural plant models to study, understand and integrate plant development and ecophysiology," *Annals of Botany*, vol. 108, no. 6, pp. 987-989, Oct 2011.
[28] 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*, vol. 35, no. 9-10, pp. 751-760, 2008.
[29] N. Chomsky, "Three Models for the Description of Language," *IEEE Transactions on Information Theory*, vol. 2, no. 3, pp. 113-124, 1956.
[30] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, no. 2, pp. 137-167, 1959.
[31] W. Petersen. (2006, 10 April 2019). *Introduction to the Theory of Formal Languages* [PDF]. Available: https://user.phil.hhu.de/~petersen/Riga/print_Folien_Riga_NLT.pdf
[32] A. Lindenmayer, "Mathematical Models for Cellular Interactions in Development .I. Filaments with 1-Sided Inputs," *Journal of Theoretical Biology*, vol. 18, no. 3, pp. 280-299, 1968.
[33] A. Lindenmayer, "Mathematical Models for Cellular Interactions in Development .2. Simple and Branching Filaments with 2-Sided Inputs," *Journal of Theoretical Biology*, vol. 18, no. 3, pp. 300-315, 1968.
[34] A. Lindenmayer, "Developmental systems without cellular interactions, their languages and grammars," *Journal of Theoretical Biology*, vol. 30, no. 3, pp. 455-484, 1971.
[35] D. van Dalen, "A note on some systems of Lindenmayer," *Mathematical Systems Theory*, vol. 5, no. 2, pp. 128-140, 1971.
[36] G. Rozenberg and P. G. Doucet, "On 0L-Languages," *Information and Control*, vol. 19, no. 4, pp. 302-318, 1971.
[37] G. T. Herman, "Computing ability of a developmental model for filamentous organisms," *Journal of Theoretical Biology*, vol. 25, no. 3, pp. 421-435, 1969.

[38] W. A. O. Feurzeig, "Programming-Languages as a Conceptual Framework for Teaching Mathematics. Final Report on the First Fifteen Months of the LOGO Project," Bolt, Beranek and Newman, Inc., Cambridge, MAR-1889, 1969.
[39] W. Feurzeig and G. Lukas, "LOGO-A Programming Language for Teaching Mathematics," *Educational Technology*, vol. 12, no. 3, pp. 39-46, 1972.
[40] P. Prusinkiewicz, "Graphical applications of L-systems," presented at the Proceedings on Graphics Interface '86/Vision Interface '86, Vancouver, British Columbia, Canada, 1986.
[41] R. Mech, P. Prusinkiewicz, and J. Hanan, "Extensions to the graphical interpretation of L-systems based on turtle geometry," Unpublished Report, 1997.
[42] H. Abelson and A. A. diSessa, *Turtle geometry : the computer as a medium for exploring mathematics*. Cambridge: MIT Press, 1980.
[43] M. Gardner, "Mathematical games: An array of problems that can be solved with elementary mathematical techniques," *Scientific American*, vol. 216, no. 3, pp. 124-129, 1967.
[44] A. Lindenmayer and P. Prusinkiewicz, "Developmental Models of Multicellular Organisms: A Computer Graphics Perspective," in *Proceedings of the Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems (ALIFE '87)*, Los Alamos, NM, USA, 1987, pp. 221-250.
[45] P. Prusinkiewicz, A. Lindenmayer, and J. Hanan, "Developmental models of herbaceous plants for computer imagery purposes," *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 141-150, 1988.
[46] F. P. Preparata and R. T.-Y. Yeh, *Introduction to Discrete Structures for Computer Science and Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1973, p. 354.
[47] A. Lindenmayer, "Adding continuous components to L-systems," in *L Systems*, G. Rozenberg and A. Salomaa, Eds. Berlin, Heidelberg: Springer, 1974, pp. 53-68.
[48] J. S. Hanan, "Parametric L-systems and their application to the modelling and visualization of plants," The University of Regina (Canada), 1992.
[49] A. J. Hanson, "Geometry for n-dimensional graphics," *Graphics Gems IV*, vol. 443, pp. 149-170, 1994.
[50] P. Schneider and D. H. Eberly, *Geometric Tools for Computer Graphics*. Elsevier, 2002.
[51] I. E. Sutherland, "Three-dimensional data input by tablet," *Proceedings of the IEEE*, vol. 62, no. 4, pp. 453-461, 1974.
[52] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips, *Introduction to Computer Graphics*. Addison-Wesley, Reading, 1994.
[53] D. Hearn, M. P. Baker, and M. P. Baker, *Computer Graphics With OpenGL*. Pearson Prentice Hall Upper Saddle River, NJ, 2004.
[54] A. Kaufman, D. Cohen, and R. Yagel, "Volume graphics," *Computer*, vol. 26, no. 7, pp. 51-64, 1993.
[55] C. Pradal, F. Boudon, C. Nouguier, J. Chopard, and C. Godin, "PlantGL: A Python-based geometric library for 3D plant modelling at different scales," *Graphical Models*, vol. 71, no. 1-6, pp. 1-21, 2009.
[56] 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*. Göttingen, Germany: Research Center Forest Ecosystems of the University of Göttingen, vol. B38, 1994, p. 192.
[57] D. Da Silva, F. Boudon, C. Godin, and H. Sinoquet, "Multiscale framework for modeling and analyzing light interception by trees," *Multiscale Modeling & Simulation*, vol. 7, no. 2, pp. 910-933, 2008.
[58] O. Kniemeyer, "Rule-based modelling with the XL/GroIMP software," in *The Logic of Artificial Life: Abstracting and Synthesizing the Principles of Living Systems; Proceedings of the 6th German Workshop on Artificial Life, April 14-16, 2004, Bamberg, Germany*, 2004, p. 56: IOS Press.

[59] A. V. Aho, J. E. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., 1983, p. 427.
[60] Y. Weimin and W. Weimin, *Data Structure: C Language Edition*, 1st ed. Bejing: Tsinghua University Press, 2002, p. 334.
[61] G. Simsion and G. Witt, *Data Modeling Essentials*. Morgan Kaufmann Publishers Inc., 2004, p. 560.
[62] R. Hirschheim, H. K. Klein, and K. Lyytinen, *Information systems development and data modeling: conceptual and philosophical foundations*. Cambridge University Press, 1995.
[63] P. P.-S. Chen, "The entity-relationship model--toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9-36, 1976.
[64] S. Shixuan and W. Shan, *Introduction to Database Systems*, 3th ed. Beijing: Higher Education Press, 2000, p. 461.
[65] J. D. Ullman, *Principles of Database Systems*, 2nd ed. New York, NY, USA: W. H. Freeman & Co., 1983, p. 484.
[66] J. D. Ullman, *Principles of Database and Knowledge - Base Systems, Vol. I*. New York, NY, USA: Computer Science Press, Inc., 1988, p. 631.
[67] D. C. Tsichritzis and F. H. Lochovsky, "Hierarchical Data-Base Management: A Survey," *ACM Comput. Surv.*, vol. 8, no. 1, pp. 105-123, 1976.
[68] J. P. Fry and E. H. Sibley, "Evolution of Data-Base Management Systems," *ACM Comput. Surv.*, vol. 8, no. 1, pp. 7-42, 1976.
[69] P. S. Strauss and R. Carey, "An object-oriented 3D graphics toolkit," *SIGGRAPH Comput. Graph.*, vol. 26, no. 2, pp. 341-349, 1992.
[70] P. S. Strauss, "IRIS Inventor, a 3D graphics toolkit," *SIGPLAN Not.*, vol. 28, no. 10, pp. 192-200, 1993.
[71] J. Rohlf and J. Helman, "IRIS performer: a high performance multiprocessing toolkit for real-time 3D graphics," in *the 21st annual conference on Computer graphics and interactive techniques*, 1994, pp. 381-394, 192262: ACM.
[72] G. Falk, "Interpretation of imperfect line data as a three-dimensional scene," *Artificial Intelligence*, vol. 3, pp. 101-144, 1972.
[73] N. Badler and R. Bajcsy, "Three-dimensional representations for computer graphics and computer vision," in *ACM SIGGRAPH Computer Graphics*, 1978, vol. 12, no. 3, pp. 153-160: ACM.
[74] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, Inc., 2015, p. 238.
[75] M. A. Rodriguez and P. Neubauer, "Constructions from dots and lines," *Bulletin of the American Society for Information Science and Technology*, vol. 36, no. 6, pp. 35-41, 2010.
[76] D. Alocci, J. Mariethoz, O. Horlacher, J. T. Bolleman, M. P. Campbell, and F. Lisacek, "Property Graph vs RDF Triple Store: A Comparison on Glycan Substructure Search," *PLOS ONE*, vol. 10, no. 12, p. 17, 2015.
[77] J. J. Miller, "Graph database applications and concepts with Neo4j," in *Southern Association for Information Systems Conference*, Atlanta, GA, USA, 2013, pp. 135-140.
[78] T. M. Schorsch and D. A. Cook, "Evolutionary Trends of Programming Languages," *The Journal of Defense Software Engineering*, pp. 4-9, Feb. 2003.
[79] P. Prusinkiewicz, J. Hanan, and R. Měch, "An L-System-Based Plant Modeling Language," Berlin, Heidelberg, 2000, pp. 395-410: Springer.
[80] M. James, J. Hanan, and P. Prusinkiewicz, "CPFG version 2.0 user's manual," Unpublished User's Manual, 1993.
[81] R. Mech, M. James, M. Hammel, J. Hanan, and P. Prusinkiewicz. (2004, 10 April 2019). *CPFG version 4.0 user's manual* [PDF]. Available: http://algorithmicbotany.org/lstudio/CPFGman.pdf

[82] R. Karwowski and P. Prusinkiewicz, "Design and Implementation of the L+C Modeling Language," *Electronic Notes in Theoretical Computer Science*, vol. 86, no. 2, pp. 134-152, 2003.
[83] P. Prusinkiewicz and R. Karwowski, "The L+C Plant-Modelling Language," *Wageningen UR Frontis Series*, vol. 22 Functional-Structural Plant Modelling in Crop Production, pp. 27-42, 2007.
[84] R. Karwowski and B. Lane. (2006, 10 April 2019). *LPFG user's manual* [PDF]. Available: http://algorithmicbotany.org/lstudio/LPFGman.pdf
[85] P. Prusinkiewicz, "Art and Science for Life: Designing and Growing. Virtual Plants with L-systems," in *XXVI International Horticultural Congress: Nursery Crops; Development, Evaluation, Production and Use*, 2004, pp. 15-28: International Society for Horticultural Science (ISHS), Leuven, Belgium.
[86] R. Karwowski and P. Prusinkiewicz, "The L-system-based plant-modeling environment L-studio 4.0," in *Proceedings of the 4th International Workshop on Functional-Structural Plant Models*, 2004, pp. 403-405: UMR AMAP Montpellier, France.
[87] O. Kniemeyer, G. H. Buck-Sorlin, and W. Kurth, "A graph grammar approach to artificial life," *Artificial Life*, vol. 10, no. 4, pp. 413-431, 2004.
[88] W. Kurth, "Specification of morphological models with L-systems and relational growth grammars," *Image–Journal of Interdisciplinary Image Science*, vol. 5, no. 1, pp. 50-79, 2007.
[89] 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*, vol. 35, no. 10, pp. 739-750, 2008.
[90] O. Kniemeyer, G. Buck-Sorlin, and W. Kurth, "GroIMP as a platform for functional-structural modelling of plants," *Wageningen UR Frontis Series*, vol. 22 Functional-Structural Plant Modelling in Crop Production, pp. 43-52, 2007.
[91] O. Kniemeyer, "Design and implementation of a graph grammar based language for functional-structural plant modelling," PhD Thesis, Brandenburg University of Technology, Cottbus, Germany, 2008.
[92] W. Kurth, O. Kniemeyer, and G. Buck-Sorlin, "Relational growth grammars - A graph rewriting approach to dynamical systems with a dynamical structure," *Unconventional Programming Paradigms*, LNCS, vol. 3566, pp. 56-72, 2005.
[93] 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*, Methods vol. 3, no. 76, 2012.
[94] F. Boudon, T. Cokelaer, C. Pradal, and C. Godin. (2017, 11 Oct 2018). *Lpy User Guide*. Available: https://lpy.readthedocs.io/en/latest/index.html
[95] F. Boudon, T. Cokelaer, C. Pradal, and C. Godin, "L-Py, an open L-systems framework in Python," in *6th International Workshop on Functional-Structural Plant Models*, 2010, pp. 116-119.
[96] F. Boudon, T. Cokelaer, C. Pradal, and C. Godin. (2017, 10 April 2019). *LPy Documentation* [PDF]. Available: https://media.readthedocs.org/pdf/lpy/develop/lpy.pdf
[97] C. Pradal, C. Fournier, P. Valduriez, and S. Cohen-Boulakia, "OpenAlea: scientific workflows combining data analysis and simulation," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, 2015, p. 11: ACM.
[98] C. Pradal, S. Dufour-Kowalski, F. Boudon, and N. Dones, "The architecture of OpenAlea: A visual programming and component based software for plant modeling," in *5th International Workshop on Functional and Structural Plant Models*, Napier, Zealand, 2007, no. 25, pp. 1-4.
[99] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation*. Worldscientific, 1997.

[100] C. Godin and Y. Caraglio, "A Multiscale Model of Plant Topological Structures," *Journal of Theoretical Biology*, vol. 191, no. 1, pp. 1-46, 1998.
[101] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.
[102] R. Gallardo, S. Hommel, S. Kannan, J. Gordon, and S. B. Zakhour, *The Java Tutorial: A Short Course on the Basics*, 6th ed. Addison-Wesley Professional, 2014, p. 864.
[103] Oracle Corporation. (2018, 11 Oct 2018). *Java Platform Standard Edition 8 Documentation*. Available: https://docs.oracle.com/javase/8/docs/
[104] M. F. Sanner, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57-61, 1999.
[105] Python Software Foundation. (2018, 11 Oct 2018). *Python 2.7.15 documentation*. Available: https://docs.python.org/2/index.html
[106] Y. Ong, "Extension of the Rule-Based Programming Language XL by Concepts for Multi-Scaled Modelling and Level-of-Detail Visualization," PhD thesis, Georg-August University School of Science (GAUSS), University of Göttingen, 2015.
[107] T. Cokelaer and C. Pradal. (2013, 11 Oct 2018). *Openalea mtg documentation*. Available: http://openalea.gforge.inria.fr/doc/vplants/newmtg/doc/_build/html/contents.html
[108] C. Godin, E. Costes, and H. Sinoquet, "A method for describing plant architecture which integrates topology and geometry," *Annals of Botany*, vol. 84, no. 3, pp. 343-357, 1999.
[109] O. Kniemeyer, M. Henke, Y. Ong, R. Hemmerling, and Q. Long. (2017, 11 Oct 2018). *GroIMP Source Code*. Available: https://sourceforge.net/p/groimp/code/HEAD/tree/trunk/
[110] F. Boudon, C. Pradal, and C. Nouguier. (2011, 11 Oct 2018). *VPlants PlantGL documentation*. Available: http://openalea.gforge.inria.fr/doc/vplants/PlantGL/doc/_build/html/contents.html
[111] B. Randell, "Software engineering in 1968," in *4th International Conference on Software Engineering*, Munich, Germany, 1979, pp. 1-10: IEEE Press.
[112] A. Brennecke and R. Keil-Slawik, "History of Software Engineering," in *Position Papers for Dagstuhl Seminar*, 1996, vol. 9635.
[113] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131-183, 1992.
[114] B. Boehm, "A view of 20th and 21st century software engineering," in *28th international conference on Software engineering*, Shanghai, China, 2006, pp. 12-29, 1134288: ACM.
[115] J. Huang and H.-g. Zhao, "Software Reuse, Software Composition and Software Integration," (in Chinese), *Application Research of Computers*, vol. 21, no. 9, pp. 118-120, 2004.
[116] W. Hasselbring, "Information system integration," *Commun. ACM*, vol. 43, no. 6, pp. 32-38, 2000.
[117] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid services for distributed system integration," *Computer*, no. 6, pp. 37-46, 2002.
[118] R. Land and I. Crnkovic, "Existing approaches to software integration–and a challenge for the future," *Integration*, vol. 40, pp. 58-104, 2004.
[119] P. A. Bernstein, "Middleware: a model for distributed system services," *Commun. ACM*, vol. 39, no. 2, pp. 86-98, 1996.
[120] R. E. Schantz and D. C. Schmidt, "Middleware for distributed systems: Evolving the common structure for network-centric applications," *Encyclopedia of Software Engineering*, vol. 1, pp. 1-9, 2001.
[121] R. Klischewski, "Information Integration or Process Integration? How to Achieve Interoperability in Administration," Berlin, Heidelberg, 2004, pp. 57-65: Springer.
[122] T. J. Mowbray and R. Zahavi, *The essential CORBA: systems integration using distributed objects*. Wiley New York, 1995.
[123] Audacia. (10 April 2019). *SYSTEMS INTEGRATION APPROACHES* [PDF]. Available: https://www.audacia.co.uk/media/1137/audacia_whitepaper_integration.pdf

[124] C. T. Howie, J. C. Kunz, and K. H. Law, "Software interoperability," Center for Integrated Facility Engineering, Stanford University117, 1996.
[125] S. Koussouris, F. Lampathaki, and D. Askounis. (2015, 10 April 2019). *Interoperability* [PDF]. Available: http://academics.epu.ntua.gr/LinkClick.aspx?fileticket=curWlwV0WY8=&tabid=385&mid=2317
[126] K. Kosanke, "ISO Standards for Interoperability: a Comparison," London, 2006, pp. 55-64: Springer.
[127] J. A. Mykkänen and M. P. Tuomainen, "An evaluation and selection framework for interoperability standards," *Information and Software Technology,* vol. 50, no. 3, pp. 176-197, 2008.
[128] Office of the National Coordinator for Health Information Technology. (2015, 10 April 2019). *Connecting health and care for the nation: A shared nationwide interoperability roadmap (1.0 ed.)* [PDF]. Available: https://www.healthit.gov/sites/default/files/hie-interoperability/nationwide-interoperability-roadmap-final-version-1.0.pdf
[129] ISO/IEC. (1993, 10 April 2019). *ISO/IEC 2382-1:1993*. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:-1:ed-3:v1:en
[130] A. Geraci, *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, 1991, p. 217.
[131] D. Hartzband. (14 Oct 2018). *Integration and Interoperability for HIT*. Available: https://www.rchnfoundation.org/?p=755
[132] B. Elvesæter, A. Hahn, A.-J. Berre, and T. Neple, "Towards an Interoperability Framework for Model-Driven Development of Software Systems," London, 2006, pp. 409-420: Springer.
[133] J. Braa and S. Sahay, *Integrated Health Information Architecture: Power to the Users*. Matrix, 2012.
[134] J. Kimber. (2013, 14 Oct 2018). *The difference between integration and interoperability*. Available: https://www.securityworldmarket.com/int/News/Comment-of-the-Month/the-difference-between-integration-and-interoperability#.W8Nrx9UzaiQ
[135] G. Smith. (2018, 14 Oct 2018). *Interface, Interoperability, Integration - A Quick Guide*. Available: https://www.cu.net/industrial/blog/integration-interface-interoperability
[136] J. Merklein, M. Poirier-Pocovi, G. H. Buck-Sorlin, W. Kurth, and Q. Long, "A dynamic model of xylem and phloem flux in an apple branch," presented at the 6th International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications (PMA2018), Hefei, China, Nov. 4 - Nov. 8, 2018.
[137] H. Haas. (18 Dec 2018). *WSDL 2.0: What's new?* Available: https://www.w3.org/2004/07/xml2004-hh/wsdl20-update.html
[138] V. Panos, "A Survey of Extract–Transform–Load Technology," *International Journal of Data Warehousing and Mining (IJDWM),* vol. 5, no. 3, pp. 1-27, 2009.
[139] R. Kimball and J. Caserta, *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, 2011.
[140] B. Aiken, J. Strassner, B. Carpenter, I. Foster, C. Lynch, J. Mambretti, R. Moore, and B. Teitelbaum. (2000, 22 Feb 2019). *Network Policy and Services: A Report of a Workshop on Middleware*. Available: https://tools.ietf.org/html/rfc2768
[141] A. T. Campbell, G. Coulson, and M. E. Kounavis, "Managing complexity: Middleware explained," *IT professional,* vol. 1, no. 5, pp. 22-28, 1999.
[142] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering,* vol. 37, no. 5, pp. 593-615, 2011.
[143] M. Müller, M. Balz, and M. Goedicke, "Representing Formal Component Models in OSGi," *Software Engineering,* vol. 159, pp. 45-56, 2010.

[144] U.S. Department of Defense. (10 April 2019). *Component Models* [PDF]. Available: https://dodcio.defense.gov/Portals/0/Documents/DODAF/Vol_1_Sect_7-2-1_Component_Models.pdf
[145] H. S. Yazdi and J. Lehmann. (2018, 10 April 2019). *Property Graph Databases* [PDF]. Available: https://sewiki.iai.uni-bonn.de/_media/teaching/lectures/kga/2018/03-propertygraphdatabases.pdf
[146] World Wide Web Consortium (W3C). (2014, 23 Feb 2019). *Resource Description Framework (RDF)*. Available: https://www.w3.org/RDF/
[147] D. Zwillinger, *CRC standard mathematical tables and formulas*, 33rd ed. Boca Raton, Florida, United States: CRC Press/Chapman and Hall, 2018, p. 858.
[148] F. Harary, "The number of linear, directed, rooted, and connected graphs," *Transactions of the American Mathematical Society*, vol. 78, no. 2, pp. 445-463, 1955.
[149] JSON-RPC Working Group. (2013, 10 April 2019). *JSON-RPC 2.0 Specification*. Available: https://www.jsonrpc.org/specification
[150] R. Koebler. (2013, 18 Feb 2019). *JSON-RPC 2.0 Transport: HTTP*. Available: https://www.simple-is-better.org/json-rpc/transport_http.html
[151] Internet Engineering Task Force (IETF). (2014, 10 April 2019). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. Available: https://tools.ietf.org/html/rfc7231
[152] Internet Assigned Numbers Authority (IANA). (2019, 18 Feb 2019). *Media Types*. Available: https://www.iana.org/assignments/media-types/media-types.xhtml
[153] M. Zou, "An algorithm for triangulating 3D polygons," Washington University in St. Louis, 2013.
[154] Python Software Foundation. (2019, 12 March 2019). *Python 3.7.2 documentation - getattribute (3.7.2 ed.)*. Available: https://docs.python.org/3/reference/datamodel.html?highlight=getattribute#object.__getattribute__
[155] Oracle Corporation. (2017, 13 March 2019). *The Java™ Tutorials - Getting and Setting Field Values*. Available: https://docs.oracle.com/javase/tutorial/reflect/member/fieldValues.html
[156] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., 2003, p. 432.
[157] C. B. Barber, D. P. Dobkin, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Trans. Math. Softw.*, vol. 22, no. 4, pp. 469-483, 1996.
[158] P. H. Cournède, Y. Chen, Q. Wu, C. Baey, and B. Bayol, "Development and Evaluation of Plant Growth Models: Methodology and Implementation in the PYGMALION platform," *Mathematical Modelling of Natural Phenomena*, vol. 8, no. 4, pp. 112-130, 2013.
[159] H. Sinoquet, X. Le Roux, B. Adam, T. Ameglio, and F. A. Daudet, "RATP: a model for simulating the spatial distribution of radiation absorption, transpiration and photosynthesis within canopies: application to an isolated tree crown," *Plant, Cell & Environment*, vol. 24, no. 4, pp. 395-406, 2001.
[160] J. Perttunen, R. Sievänen, and E. Nikinmaa, "LIGNUM: a model combining the structure and the functioning of trees," *Ecological Modelling*, vol. 108, no. 1, pp. 189-198, 1998.
[161] C. Baey, A. Didier, S. Lemaire, F. Maupas, and P.-H. Cournède, "Parametrization of five classical plant growth models applied to sugar beet and comparison of their predictive capacity on root yield and total biomass," *Ecological Modelling*, vol. 290, pp. 11-20, 2014.
[162] P. H. Cournède, V. Letort, A. Mathieu, M. Z. Kang, S. Lemaire, S. Trevezas, F. Houllier, and P. de Reffye, "Some Parameter Estimation Issues in Functional-Structural Plant Modelling," *Mathematical Modelling of Natural Phenomena*, vol. 6, no. 2, pp. 133-159, 2011.

[163] Q.-L. Wu, P.-H. Cournède, and A. Mathieu, "An efficient computational method for global sensitivity analysis and its application to tree growth modelling," *Reliability Engineering & System Safety*, vol. 107, pp. 35-43, 2012.
[164] Y. Chen and P. Cournede, "Assessment of parameter uncertainty in plant growth model identification," in *2012 IEEE 4th International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications*, 2012, pp. 85-92.
[165] K. Streit, M. Henke, B. Bayol, P. Cournède, R. Sievänen, and W. Kurth, "Impact of geometrical traits on light interception in conifers: Analysis using an FSPM for Scots pine," in *2016 IEEE International Conference on Functional-Structural Plant Growth Modeling, Simulation, Visualization and Applications (FSPMA)*, 2016, pp. 194-203.
[166] C. Pradal. (2017, 31 March 2019). *OpenAlea / Lignum interface*. Available: https://github.com/openalea/lignum
[167] C. Pradal, "OpenAlea / GroIMP," Unpublished Slides, May 29, 2015.
[168] D. Abrahams and S. Seefeld. (2018, March 31 2019). *Boost.Python*. Available: https://www.boost.org/doc/libs/1_69_0/libs/python/doc/html/index.html
[169] SWIG developers. (2018, 31 March 2019). *SWIG*. Available: http://www.swig.org/
[170] The SciPy community. (2019, 31 March 2019). *F2PY Users Guide and Reference Manual*. Available: https://docs.scipy.org/doc/numpy/f2py/
[171] F. Reyes, B. Pallas, C. Pradal, F. Vaggi, D. Zanotelli, M. Tagliavini, D. Gianelle, and E. Costes, "MuSCA: a multi-scale model to explore carbon allocation in plants," *bioRxiv*, 2018.
[172] T. Hölttä, T. Vesala, S. Sevanto, M. Perämäki, and E. Nikinmaa, "Modeling xylem and phloem water flows in trees according to cohesion theory and Münch hypothesis," *Trees*, vol. 20, no. 1, pp. 67-78, 2006.
[173] University of Göttingen. (11 April 2019). *GroIMP Documentation*. Available: http://wwwuser.gwdg.de/~groimp/grogra.de/documentation/