Przemyslaw Prusinkiewicz (1), Jim Hanan (2) and Radomir Mech (3)

(1) Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
e−mail: pwp@cpsc.ucalgary.ca

(2) CSIRO Cooperative Research Centre for Tropical Pest Management
Brisbane, Queensland, Australia

(3) SGI, Mountain View, California, U.S.A

Abstract

Cpfg is a program for simulating and visualizing plant development, based on the theory of L−systems. A special−purpose programming language, used to specify plant models, is an essential feature of cpfg. We review postulates of L−system theory that have influenced the design of this language. We then present the main constructs of this language, and evaluate it from a user's perspective.

Keywords: L−system, plant modeling, cpfg

Reference

Przemyslaw Prusinkiewicz, Jim Hanan and Radomir Mech. An L−system−based plant modeling language. In: M. Nagl, A. Schuerr and M. Muench (Eds): Applications of graph transformations with industrial relevance. Proceedings of the International workshop AGTIVE'99, Kerkrade, The Netherlands, September 1999. *Lecture Notes in Computer Science* 1779, Springer, Berlin, 2000, pp.395−410.

An L-system-based plant modeling language

Przemyslaw Prusinkiewicz[1], Jim Hanan[2] and Radomír Měch[3]

[1] Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4
[2] CSIRO Cooperative Research Centre for Tropical Pest Management, Brisbane, Queensland, Australia
[3] SGI, Mountain View, California, U.S.A.

Abstract. Cpfg is a program for simulating and visualizing plant development, based on the theory of L-systems. A special-purpose programming language, used to specify plant models, is an essential feature of cpfg. We review postulates of L-system theory that have influenced the design of this language. We then present the main constructs of this language, and evaluate it from a user's perspective.

1 Introduction

L-systems were introduced in 1968 as a mathematical theory of multi-cellular development [19, 20], but soon afterwards they began to be used as a foundation for plant modeling and simulation systems [30]. The first L-system-based plant modeling program, CELIA (an acronym for the **CE**llular **L**inear **I**terative **A**rray simulator) was created by Baker and Herman in the early seventies [2], and was improved until the mid eighties. CELIA was followed by pfg (**p**lant and **f**ractal **g**enerator) [10, 33], and its successor, cpfg (**pfg** with **c**ontinuous parameters) [11, 26, 27]. Existing and prospective applications of cpfg include computer animation and landscape design, research and education in botany and ecology, and decision support in agriculture, horticulture, and forestry [38]. The synergy between scientific and visual objectives of plant modeling has been discused in [31].

A distinctive feature of cpfg is its modeling language, which makes it possible for the user to easily specify and modify a wide range of plant models[1]. The cpfg modeling language [26] extends notions of L-system theory [12, 39] with the following concepts:

1. parameters associated with L-system symbols to express quantitative attributes of the modeled structures [11, 37],
2. programming constructs borrowed from other programming languages: local and global variables, arrays, input-output functions, and flow control statements [11, 15, 34],
3. modeling constructs without obvious counterparts in other programming languages: decomposition and interpretation rules [27] and sub-L-systems [11],

[1] Related languages have been also implemented in other modeling programs based on L-systems, for example GROGRA [17] and World Builder [1].

4. programming constructs needed to capture plant responses to environmental factors and to simulate bi-directional interaction between plants and their environment [27, 28, 35],
5. graphical interpretation of L-systems based on turtle geometry [37].

In addition, the language supports graphical modeling and rendering techniques needed for realistic visualization of the models: predefined bicubic surfaces for representing plant organs of a given shape [10, 37], developmental bicubic surfaces for animating organ development [11], generalized cylinders for modeling stems with arbitrary cross-sections, and texture-mapped surfaces needed for more realistic rendering [27].

In this paper we present the design of the cpfg modeling language. We focus on items 2 and 3 of the above list, which have not been previously published outside of dissertations and theses. We begin by reviewing the elements of L-system theory that have influenced the design of the cpfg language (Section 2). On this basis, we present its essential constructs (Section 3). We conclude by summarizing our experience with the cpfg language, and discussing areas that require further research (Section 4).

2 L-systems as a plant modeling paradigm

2.1 A plant as a metapopulation

A basic postulate of L-system theory [24, 30] is that a plant can be considered as a population (set) of discrete components, such as apices, internodes, leaves, and flowers. In simpler multicellular organisms, for example algae, these components can be identified with individual cells [19]. It is assumed that the set of organ types in organisms of a given species is finite, irrespective of the organism size. The type of an organ is represented by a symbol. Since the set of organ types is finite, the set (alphabet) V of symbols is finite as well.

2.2 Branching architecture of plants

L-system models operate at the level of plant architecture, which means that components of a model are assumed to be connected into a branching structure. From the graph-theoretic point of view, this structure can be described as an axial tree. Organs are represented by edges of this tree, and identified by symbols from alphabet V used as edge labels.

Formally, an axial tree is a special type of rooted tree [37]. At each of its nodes we distinguish at most one outgoing edge called the straight segment. All remaining edges are called lateral or side segments. Within an axial tree, a sequence of edges is called an axis if: (i) the first edge in the sequence originates at the root of the tree or as a lateral segment at some node, (ii) each subsequent edge is a straight segment, and (iii) the last edge is not followed by any straight segment. The beginning and ending node of an axis are called its base and tip, respectively. An axis with all its descendants (edges that can be reached from the nodes within this axis) is called a branch.

2.3 Development as a parallel rewriting process

According to the theory of L-systems, plant development can be captured by a set of productions that describe the fate of plant components over discrete time intervals, beginning with an initial structure (the axiom) [24, 30]. In graph-theoretic terms, a production replaces an edge of an axial tree, called the production predecessor, by an axial subtree called the successor. In the simplest case of development controlled by lineage [23, 32, 37], the suitable production is identified by the label of its predecessor, which must match the label of the replaced edge in the tree. The successor is embedded into the resulting tree in such a manner that the starting node of the predecessor edge is mapped to the base of the successor axis, and the end node of the predecessor edge is mapped to the tip of the successor axis. Productions replace all modules of the predecessor tree in parallel derivation steps. This parallelism is intended to reflect simultaneous progress of time in all parts of the modeled organism. For example, Figure 1 presents the development of a hypothetical compound leaf with two segment types: the apices (thin lines) and the internodes (thick lines). The development begins with a single apical segment and is modeled using two productions.

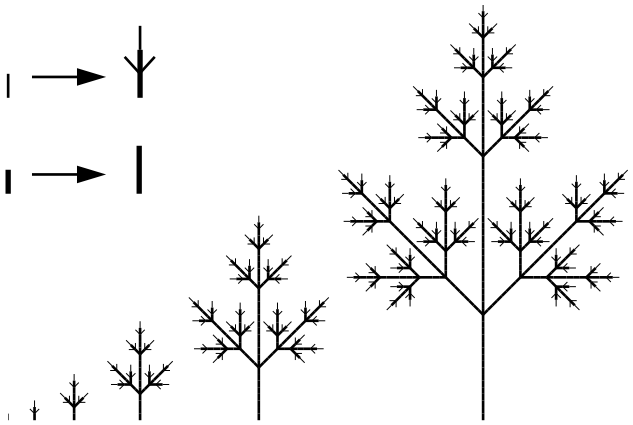

Fig. 1. Productions of a sample L-systems and a sequence of derived structures

2.4 The bracketed string notation

Within the theory of L-systems, axial trees are commonly specified as strings of symbols (words) over the alphabet $V \cup \{[,]\}$, where the bracket symbols $[,]$ do not belong to the set of component symbols V. A sequence of labels of consecutive straight segments represents an axis. A matching pair of brackets $[\ldots]$ encloses a branch. For example, let A and B be symbols from alphabet V, and w be a properly bracketed string with symbols from V. The notation $\ldots A[w]B\ldots$ means that B is the straight segment that follows A in the axis $\ldots AB\ldots$, and w is the lateral branch originating at the end node of A.

Every axial tree can be described by a well-formed bracketed string, and every well-formed bracketed string describes an axial tree (*cf.* [36]). Consequently, bracketed string notation provides a convenient means for specifying an L-system (the axiom and the set of productions) in a textual form. For example, the L-system shown in Figure 1 can be written as

$$\begin{array}{l} A \\ A \rightarrow I[+A][-A]IA \\ I \rightarrow II \end{array} \tag{1}$$

where A denotes an apex and I denotes an internode segment. Symbols + and − indicate the directions of branching (to the left and to the right, respectively) according to the turtle geometry paradigm [37].

2.5 Parametric L-systems

It is often necessary to characterize components of the modeled structure using continuously valued parameters. For example, parameter values may represent geometrical aspects of components, such as the length and diameter of an internode, relationships between components, such as the magnitude of a branching angle, and physiological attributes of a component, such as its water content or concentration of photosynthates. The association of numerical parameters with L-system symbols was first described by Lindenmayer [21]. The cpfg language is based on a formalization of this concept called parametric L-systems [11, 37].

Parametric L-systems operate on bracketed parametric strings, that is strings of modules defined as symbols with the associated numerical (real-valued) parameters. The actual parameters that appear in parametric strings have their counterparts in formal parameters that may appear in productions. For example, a production that doubles length x of an internode I in every derivation step may be written as

$$I(x) \rightarrow I(2 * x). \tag{2}$$

The above concepts serve as the foundation of the cpfg modeling language, described next.

3 The cpfg modeling language

3.1 Example of a simple cpfg model

A simple cpfg model can be specified by complementing the axiom (listed after the keyword `axiom`) and productions of a parametric L-system with three statements: `lsystem:` *label*, `endlsystem`, and `derivation length:` *length*. The first two statements delimit the L-system and assign a unique label to it; this makes it possible to divide more complex models into several sub-L-systems (Section 3.5). The remaining statement specifies the required derivation length. For example, a cpfg model describing the development of the compound leaf shown in Figure 1 may be written as follows:

Model 1

```
#define growth_rate 2
lsystem: 1
derivation length: 5
axiom: A
A --> I(1)[+A][-A]I(1)A
I(x) --> I(growth_rate * x)
endlsystem
```

This model combines the basic structure of L-system (1) with the parametric specification of internode elongation by production (2). The use of a parameter makes it possible to avoid the exponential increase of the number of symbols representing a growing internode, and makes it easy to modify the internode growth rate. This is emphasized by the preprocessed `#define` statement, which assigns a value to the `growth_rate` constant.

3.2 Productions, variables, and statement blocks

Model 1 incorporates the simplest type of productions used in cpfg models, the deterministic context-free productions. Cpfg also supports context-sensitive productions [19, 23], which make it possible to capture a wide range of interactions between components of a branching structure [32]. Overall, cpfg productions have the following syntax [11]:

$$lc \texttt{ < } pred \texttt{ > } rc \texttt{ : \{ } \alpha \texttt{ \} } cond \texttt{ \{ } \beta \texttt{ \} --> } succ \texttt{ : } prob. \tag{3}$$

The terms *lc*, *pred*, *rc*, and *succ* are parametric strings denoting the left context, the strict predecessor, the right context, and the successor of the production. The strict predecessor must not be the empty string. It usually consists of a single module, but may also include several modules. The strict predecessor and the successor, separated by an arrow, are the only mandatory terms of a production; all other terms and the separators related to them can be omitted.

Cpfg productions can be applied in a deterministic or stochastic fashion. In the deterministic case, productions in the model are scanned in the order in which they appear, and the first applicable production is used. In contrast, in the stochastic case (indicated by the presence of *prob* expressions at the end of production specifications) all applicable productions are found, and one of them is selected at random. Specifically [32], if $p_{i_1}, p_{i_2}, \ldots, p_{i_m}$ are the applicable productions, and $\pi_{i_1}, \pi_{i_2}, \ldots, \pi_{i_m} \geq 0$ are the corresponding values of their *prob* expressions, a production p_{i_k} will be selected with the probability

$$P(p_{i_k}) = \frac{\pi_{i_k}}{\sum_{v=1}^{m} \pi_{i_v}}, \qquad k = 1, 2, \ldots, m. \tag{4}$$

The condition *cond* is a logical expression that guards the application of the production (the production may only be applied when *cond* evaluates to *true*). The term α is a block of statements that is executed before the evaluation of

the condition *cond*. Similarly, β is a block of statements that is executed after the condition evaluation, if the result is *true*. The condition and the statement blocks α and β are expressed using a syntax based on the C programming language [16]. Arithmetic expressions may also be included in the successor specification, where they define the parameter values assigned to the successor modules as discussed in Section 2.5. The logical and arithmetic expressions may include the usual logical operators (Boolean operations AND, OR, NOT, and comparisons of numerical values), arithmetic operators (addition, subtraction, multiplication, division, remainder from integer division, and raising to a power), and predefined functions. The supported functions include a selection of standard mathematical functions (*e.g.* `sin`, `atan`, `exp`, `floor`, `sign`) and functions that return pseudo-random values with given distributions (uniform, normal, beta). In addition, the statement blocks may include assignments, `if` and `if ... else` statements, `while` and `do ... while` loops, and C-like input-output functions (`printf`, `fopen`, `fclose`, `fprintf`, `fscanf`).

The expressions and statements included in productions operate on formal arguments listed in the production predecessor and context, local variables (introduced in blocks α or β, and with a scope limited to individual productions) and global variables, which can be accessed by all productions. A global variable must be initialized in one of the following statement blocks:

`Start:` {*statements*}	(executed at the beginning of the simulation),
`End:` {*statements*}	(executed at the end of the simulation),
`StartEach:` {*statements*}	(executed at the beginning of each step),
`EndEach:` {*statements*}	(executed at the end of each step).

The statement blocks may be specified in any order between the `lsystem` and `axiom` statements. In addition to global variables, the cpfg language also supports globally defined arrays.

The use of these constructs is illustrated by the following model:

Model 2

```
#define dt    0.03    /* time increment */
#define t_max 1.0     /* maximum age of the apex */
lsystem: 1
Start: {fp = fopen("statistics", "w"); step=0;}
StartEach: {step=step+1; n=0;}
EndEach: {fprintf(fp, "Step %f, number of apices %f\n", step, n);}
End: {fclose(fp);}
derivation length: 200
axiom: A(0)
/* p1: advance apex age until t_max */
A(t) : {t_new = t+dt;} t_new<t_max {n=n+1;} --> A(t_new)
/* p2: create new organs when t_max has been reached */
A(t) : {t_new = t+dt;} t_new>=t_max
        {t_init = t_new-t_max; n=n+3;} -->
        I(t_init)[+A(t_init)] [-A(t_init)] I(t_init) A(t_init)
```

```
/* p3: advance internode age */
I(t) --> I(t+dt)
endlsystem
```

The `Start` statements open the output file `statistics` and initialize the global variable `step` that will count derivation steps. The `StartEach` statements increment the `step` variable at the beginning of each derivation step, and set to zero a global variable `n`, used to count the total number of apices in the structure. The resulting number is reported (appended to the file `statistics`) at the end of each derivation step by the `EndEach` statement. Finally, the `End` statement closes the file `statistics` at the end of simulation.

In contrast to Model 1, in which the time increment associated with a derivation step was an inherent feature of the model, in Model 2 the time increment is controlled by a user-definable constant `dt`. According to production `p1`, the age of an apex advances by the constant `dt` until the maximum age value `t_max` has been reached. At that time the apex divides into several new modules, as described by production `p2`. Since time advances by fixed increments `dt`, the age `t_new` may actually exceed the maximum `t_max`, which is why the newly created modules are assigned the initial age `t_init = t_new - t_max` by production `p2`.

The model makes use of global variables to collect the output data that quantify the simulation results. Productions `p1` and `p2` increment the global variable `n` by the number of created apices (1 and 3, respectively). Consequently, variable `n` represents the total number of apices at the end of each derivation step. The sequence of these values constitutes the numerical output of the simulation.

Strictly speaking, the use of global variables that can be changed by productions is inconsistent with the parallel application of productions postulated by the definition of L-systems. The reason is that different productions may attempt to assign different values to the same variable at the same time [34]. We address this problem at a conceptual level by assuming that the blocks of statements associated with productions are executed as indivisible operations in an arbitrary order. Thus, we use interleaving composition as a model of parallelism when evaluating these blocks [25]. In practice, cpfg is implemented as a sequential program that applies productions one at a time (*cf.* [33, Appendix A]), thus the assumption of indivisible execution of the statement blocks is automatically satisfied.

3.3 Decomposition rules

As noted in Section 2.3, an L-system production such as $A \rightarrow BC$ states that module A produces modules B and C over time. In the context of plant modeling it is also convenient to have a construct for expressing another concept, namely that a compound module A consists of (or can be decomposed into) modules B and C. In cpfg, such structural relations are expressed using decomposition rules. They have the same syntax as context-free productions described in the previous section, but are preceded by the keyword `decomposition` in the cpfg

model. When decomposition rules are written outside a cpfg program, they are indicated by symbol `-d>` used instead of `-->` in the production specification.

A derivation step in a model with decomposition consists of the application of productions, followed immediately by the application of decomposition rules. In general, the decomposition rules can be applied recursively, as long as there are modules that can be further decomposed. This possibility is illustrated by the following variant of Model 2:

Model 3

```
#define dt 1.3 /* time increment */
#define t_max 1.0
lsystem: 1
derivation length: 0
axiom: A(0)
A(t) --> A(t+dt)
I(t) --> I(t+dt)
decomposition
A(t) : t>=t_max {t_init = t-t_max;} --> M(t_init ) A(t_init)
M(t) --> I(t) [+A(t)] [-A(t)] I(t)
endlsystem
```

In Model 3, productions simply increment the age of apices `A` and internodes `I` in each derivation step. The fate of an apex that has reached its maximum age `t_max` is expressed by the decomposition rules. The first rule states that such an apex will produce a compound structure `M` (a metamer [3]) and a younger instance of the apex `A`. The second rule specifies that `M` consists of two internode segments `I`, which support a pair of lateral apices `A` at their join point. Thus, the description of the periodic activity of the apex has been separated from the detailed description of the produced structure `M`. The first derivation step in Model 3 is illustrated in Figure 2.

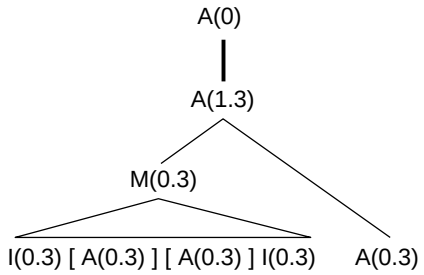

Fig. 2. Illustration of a derivation step with decomposition. The application of the production (thick line) is followed by the application of decomposition rules (thin lines).

If the time increment `dt` is greater than `t_max`, the initial age `t_init` of the newly produced apices may still be greater than `t_max`, resulting in a recursive

application of the decomposition rules. This recursion will eventually end, because the first decomposition rule reduces the age of new modules by `t_max` with respect to the age of their parents. Thus, in addition to clarifying model specification, decomposition rules make it possible to improve the models. Specifically, Model 2 operates correctly only if `dt < t_max`, whereas Model 3 does not require this assumption. The formal basis for advancing time by arbitrarily large steps using recursively applied productions was introduced in [37] (timed L-systems).

The logical distinction between the relations "produced over time" and "being part of", which underlies the distinction between productions and decomposition rules in cpfg models, was formalized by Woodger and Tarski [41], and reviewed by Lindenmayer [22]. The multi-level specification of plants, implicit in the distinction between compound modules and their constituents, was analyzed by Godin and Caraglio [8].

3.4 Interpretation rules

Using the terminology of [37], Models 1–3 are schematic: they only specify the topology of the developing structures. Interpretation rules provide a mechanism for complementing schematic models with the geometric information needed for visualization purposes. The interpretation rules do not affect the sequence

$$\begin{array}{ccccccccc} \mu_0 & \overset{P}{\Longrightarrow} & \mu_1 & \overset{P}{\Longrightarrow} & \mu_2 & \overset{P}{\Longrightarrow} & \mu_3 & \overset{P}{\Longrightarrow} & \cdots \\ \Downarrow h & & \Downarrow h & & \Downarrow h & & \Downarrow h & & \\ \nu_0 & & \nu_1 & & \nu_2 & & \nu_3 & & \cdots \end{array}$$

Fig. 3. Generation of a developmental sequence using a cpfg model with interpretation rules. The progression of strings $\mu_0, \mu_1, \mu_2, \ldots$ results from the derivation steps $\overset{P}{\Longrightarrow}$ defined by productions and decomposition rules. The interpretation rules $\overset{h}{\Longrightarrow}$ map strings μ_i into the final strings ν_i that contain the graphical information.

of strings $\mu_0, \mu_1, \mu_2, \ldots$ derived by productions and decomposition rules, but make it possible to replace modules in the derived strings by other modules or sequences of modules (Figure 3), which may have a predefined graphical interpretation. This concept was first applied to L-system-based plant modeling by Kurth [17].)

In cpfg, the interpretation rules are specified using the same syntax as context-free productions and decomposition rules, following the keyword `homomorphism`. Considered in isolation, they are indicated by the symbol `-h>` used instead of the `-->` in the production specification. The keyword `homomorphism` reminds us that, in the simplest case, the interpretation rules define a homomorphic image [12, 39] of the string that has been generated using productions and decomposition rules. In general, however, the interpretation rules extend the concept of

homomorphism, because they may operate on symbols with parameters, and can be applied in hierarchical and recursive manners. In that sense, they resemble decomposition rules.

For example, to visualize the structures generated by Models 2 or 3, one could add the following lines before the `endlsystem` statement:

```
homomorphism
A(t) --> F(t/t_max)           /* rule h1 */
I(t) --> F(0.5*2^(t/t_max))   /* rule h2 */
```

These rules replace modules `A(t)` (the apices) and `I(t)` (the internodes) with the modules `F(x)`, which are interpretated as straight line segments of length `x` [37]. According to rule `h1`, the length of an apex increases linearly with the apex age `t` and reaches 1 at the division time, `t = t_max`. According to rule `h2`, the length of an internode will double over every interval `t_max`. Due to the constant 0.5, a pair of newly created internodes will have the same combined length as the apex that created them. This guarantees that the leaf shape will change continuously with time (the model will satisfy the C^0 continuity criterion [37]). In conclusion, the interpretation rules make it possible to separate the logic of developmental programs from model visualization. The resulting models are more clearly organized and easier to understand than the models in which both aspects of specification are interwoven.

3.5 Sub-L-systems

It is convenient to apply concepts of structural programming to plant modeling and allow the modeler to partition large models into relatively independent parts. For example, a modeler may want to specify the overall development of a plant branching structure separately from the development of individual plant organs, such as leaves and inflorescences, then combine these specifications into a comprehensive plant model. Such a structured approach increases the efficiency of model design and makes it possible to reuse the same components in different models.

The partition of a developmental model into components is conceptually more difficult than the inclusion of predefined shapes into a static structure [10, 37], since the components may undergo changes as time progresses. To support the partitioning of developmental models, Hanan introduced the notion of sub-L-systems [11]. Sub-L-systems can be compared to subroutines in that they are invoked from the main L-system or other sub-L-systems to perform well defined, encapsulated tasks. Unlike sub-routines in a sequential program, however, the main-L-system and the sub-L-systems may be active at the same time. Thus, decomposition of a developmental model into the main L-system and a set of sub-L-systems preserves the parallel rewriting inherent in L-systems.

From the viewpoint of formal language theory, sub-L-systems are related to continuous grammars [6], in which the L-system-like rewriting mechanism is

applied to subwords of the rewritten word. Sub-L-systems generalize this concept by allowing productions from different sets to be applied simultaneously to specific substrings of the rewritten string in any derivation step.

In the cpfg language, separate parts of a model are identified by the numerical identifier *id* following the keyword `lsystem` (*cf.* Section 3.1). The main L-system is the first one in the model. To invoke a sub-L-system, the calling L-system produces a pair of reserved modules `$`(*id*) and `$`, which delimit the substring to be rewritten using productions of L-system *id*. These delimiters must occur in matching pairs and may be nested within the string. The module with the *id* parameter invokes the sub-L-system, while the module without parameters returns control to the higher-level set of rules (Figure 4).

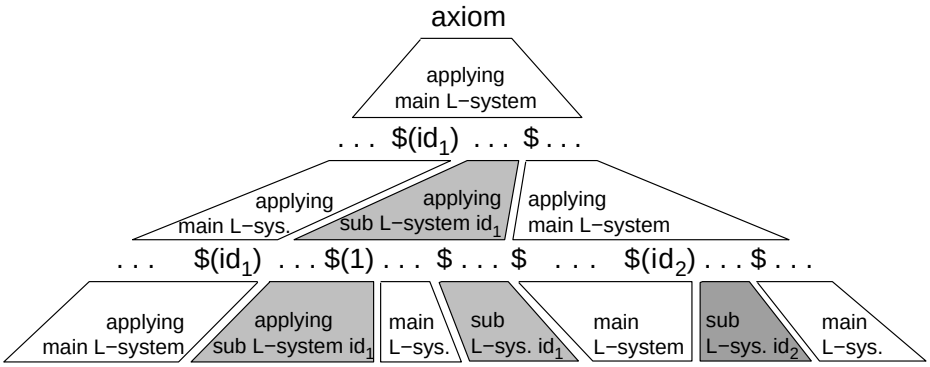

Fig. 4. Example of a developmental sequence generated by an L-system with two sub-L-systems

For example, the following schematic model makes use of the sub-L-system mechanism to separate the development of a monopodial inflorescence [37] from the development of the individual flowers.

Model 4

```
lsystem: 1            /* main L-system  */
derivation length: 3
axiom: A              /* initial string  */
A --> I[$(2)A$]A      /* production p11 */
endlsystem

lsystem: 2            /* sub-L-system   */
derivation length: 1  /* ignored        */
axiom: ABC            /* ignored        */
A --> B               /* production p21 */
B --> C               /* production p21 */
endlsystem
```

This model generates the following sequence of parametric strings:

```
A
I[$(2)A$]A
I[$(2)B$]I[$(2)A$]A
I[$(2)C$]I[$(2)B$]I[$(2)A$]A}
```

In each step, the production that is applied to an apex `A` depends on the sub-L-system identified by the delimiters immediately enclosing it. Production `p11` is applied to the apex `A` at the right end of each string, creating an internode `I`, a lateral branch incorporating the sub-L-system reference `$(2)A$`, and a new apex `A`. Production `p21` is applied to the module `A` appearing in the newly created branch, producing a blossom `B`. In the next step, production `p22` will transform this blossoms into a fruit `C`. The axiom and the derivation length of the sub-L-system do not affect the simulation, but are needed when the sub-L-system is being developed and tested independently of the main L-system.

4 Evaluation and conclusions

Consecutive versions of cpfg and its predecessor pfg have been developed and used over the last 15 years to support research in plant modeling, computer graphics, and botany. This long life span of cpfg results, first of all, from the soundness of the L-system theory that underlies its design. The L-system-based modeling language described in this paper is the essential feature of cpfg and provides the following benefits [28]:

- At the conceptual level, it facilitates the design, specification, documentation, and comparison of models.
- At the level of model implementation, it makes it possible to develop software that can be reused in various models. Specifically, graphical capabilities needed to visualize the models become a part of the modeling program (cpfg), and do not have to be reimplemented.
- Finally, the language facilitates interactive experimentation with the models.

The cpfg modeling language incorporates many constructs that extend the notion of L-systems as used in formal language theory. Global variables and C-like functions make it possible to input experimental data to the models and output simulation results for further statistical analysis. Decomposition, interpretation rules, and sub-L-systems lead to conceptually clear and well structured model specifications. These capabilities play an essential role in the applications of cpfg to biological research and image synthesis.

The cpfg modeling language inherits the conciseness of the mathematical notation of L-systems on which it is based. The user can specify simple models with only a few lines of code, and modify them easily during experimentation. The concise notation also emphasizes the conceptually intriguing database amplification property inherent in many L-system models, *i.e.,* the contrast between

the short specification of the models and the intricacy of the resulting structures and behaviors [40].

Our experience with cpfg, and that of other users, has also highlighted shortcomings of the present cpfg language. To begin with, a down side of the concise notation is that large cpfg models look cryptic. The challenge is thus to define a more legible language based on L-systems. It should improve the clarity of specification, documentation, and ease of maintenance of complex models, while still allowing for compact specification of simple models. The first step towards this goal might be the incorporation of additional constructs found in other programming languages. The most needed ones appear to be: multi-letter module type identifiers providing an alternative to one-letter symbols, user-definable functions, and user-definable data structures that could be passed as parameters to modules[2]. The data structures would eliminate the need for the long parameter lists that currently must be included in each reference to a cpfg module with many parameters. This goal can also be achieved using name-value pairs, as suggested by Borovikov [4].

The current format of production specification also could be improved. In many models the same predecessor module yields different successors depending on the context and the logical conditions that guard production application. According to the cpfg syntax, specification of different successors requires the use of separate productions. Thus, the same statement block α may have to be specified and executed several times, separately for each production, in order to provide arguments to different conditional expressions *cond* (*cf.* Section 3.2). An alternative is to introduce a more flexible production format that would allow for the selection of one of several successors depending on the production's context and conditions. A sample production syntax satisfying that requirement was proposed by Hammel [9].

The impact of using the bracketed string notation to specify productions operating on axial trees should also be re-examined. For example, strings $A[B][C]D$ and $A[C][B]D$ represent the same tree, yet the context-sensitive production $A > [B] \rightarrow X$ will only apply to the first representation. Ideally, the result of production application should not depend on the form of the string representing a given tree. Furthermore, an axial tree may have an indeterminate number of branches attached to the same node, but the bracketed string notation only makes it possible to specify a finite number of them as a production context. Consequently, concepts such as "a signal coming from any branch" or "signals coming from all branches" cannot be expressed, at present. One approach to address these problems may be to consider the rewriting of trees as a special case of a graph rewriting mechanism rather than a generalization of string rewriting, and examine the notions of graph L-systems [5, 29] and parallel graph grammars [7] as a possible foundation of an alternative plant modeling language.

Apart from the practically motivated improvements to the cpfg language, an interesting problem is the characterization of L-system-based languages in

[2] Multi-letter module names and simple function definitions can be introduced in the current cpfg models using a macro preprocessor.

relation to other programming languages. Two of the observed relations are listed below.

- A production, for instance $F(x) \rightarrow F(2*x)$, can be read as follows: "If a module is of type F and the value of its parameter is x, then it will be replaced by a module of the same type F, with the parameter value multiplied by two." Thus, a production is a declarative statement, and a cpfg model consists of a set of such statements. This relates the cpfg language to the declarative style of programming found, in particular, in logic programming languages. The relation between L-systems and declarative programming was first studied by Lewis, and led to a concise implementation of the L-system derivation mechanism in Prolog [18]. The declarative aspects of L-system models still require an in-depth analysis.
- An L-system derivation step captures time advancement by some interval. The time component inherent in production application relates L-systems to simulation languages. This relation was partially explored by Hogeweg [13, 14] and Hammel [9], who implemented L-system models in Simula.

Over the years, the notion of L-systems led to the development of an entire branch of formal language theory, and became an inherent component of architectural plant modeling. We expect that further research will lead to an improved design of practical plant modeling languages based on L-systems, and a better understanding of the place of L-systems in programming language theory.

Acknowledgments

We wish to acknowledge Christophe Godin for co-authoring the concept of decomposition rules. This research has been supported in part by research and equipment grants from the Natural Sciences and Engineering Research Council of Canada.

References

1. AnimaTek, Inc. AnimaTek's World Builder. PC program, 1996,1998.
2. R. Baker and G. T. Herman. Simulation of organisms using a developmental model, parts I and II. *International Journal of Bio-Medical Computing*, 3:201–215 and 251–267, 1972.
3. A. Bell. *Plant form: An illustrated guide to flowering plants.* Oxford University Press, Oxford, 1991.
4. I. A. Borovikov. L-systems with inheritance: an object-oriented extension of L-systems. *ACM SIGPLAN Notices*, 30(5):43–60, 1995.
5. K. Culik II and A. Lindenmayer. Parallel graph generating and graph recurrence systems for multicellular development. *Int. J. General Systems*, 3:53–66, 1976.
6. A. Ehrenfeucht, H. Maurer, and G Rozenberg. Continuous grammars. *Information and Control*, 46:71–91, 1980.
7. H. Ehrig and H.-J. Kreowski. Parallel graph grammars. In A. Lindenmayer and G. Rozenberg, editors, *Automata, languages, development*, pages 425–442. North-Holland, Amsterdam, 1976.

8. C. Godin and Y. Caraglio. A multiscale model of plant topological structures. *Journal of Theoretical Biology*, 191:1–46, 1998.
9. M. Hammel. *Differential L-systems and their application to the simulation and visualization of plant development.* PhD thesis, University of Calgary, June 1996.
10. J. S. Hanan. *PLANTWORKS: A software system for realistic plant modelling.* Master's thesis, University of Regina, November 1988.
11. J. S. Hanan. *Parametric L-systems and their application to the modelling and visualization of plants.* PhD thesis, University of Regina, June 1992.
12. G. T. Herman and G. Rozenberg. *Developmental systems and languages.* North-Holland, Amsterdam, 1975.
13. P. Hogeweg. Simulating the growth of cellular forms. *Simulation*, pages 90–96, September 1978.
14. P. Hogeweg. Locally synchronized developmental systems: Conceptual advantages of discrete event formalism. *International Journal of General Systems*, 6:57–73, 1980.
15. M. James, J. Hanan, and P. Prusinkiewicz. CPFG version 2.0 user's manual. Manuscript, Department of Computer Science, University of Calgary, 1993, 50 pages.
16. B. W. Kernighan and D. M. Ritchie. *The C programming language. Second edition.* Prentice Hall, Englewood Cliffs, 1988.
17. 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 modeling. Introduction and reference manual.* Forschungszentrum Waldökosysteme der Universität Göttingen, Göttingen, 1994.
18. P. Lewis. *A botanical plant modeling system for remote sensing simulation studies.* PhD thesis, Universiy of London, 1996.
19. A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.
20. A. Lindenmayer. Developmental systems without cellular interaction, their languages and grammars. *Journal of Theoretical Biology*, 30:455–484, 1971.
21. A. Lindenmayer. Adding continuous components to L-systems. In G. Rozenberg and A. Salomaa, editors, *L Systems*, Lecture Notes in Computer Science 15, pages 53–68. Springer-Verlag, Berlin, 1974.
22. A. Lindenmayer. Theories and observations of developmental biology. In R. E. Butts and J. Hintikka, editors, *Foundational problems in special sciences*, pages 103–118. D. Reidel Publ. Co, Dordrecht-Holland, 1977.
23. A. Lindenmayer. Developmental algorithms: Lineage versus interactive control mechanisms. In S. Subtelny and P. B. Green, editors, *Developmental order: Its origin and regulation*, pages 219–245. Alan R. Liss, New York, 1982.
24. A. Lindenmayer and H. Jürgensen. Grammars of development: Discrete-state models for growth, differentiation and gene expression in modular organisms. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 3–21. Springer-Verlag, Berlin, 1992.
25. H. McEvoy. *Coordinating multiset transformers.* PhD thesis, University of Amsterdam, October 1997.
26. R. Měch. CPFG version 3.4 user's manual. Department of Computer Science, University of Calgary, May 1998.

27. R. Měch. *Modeling and simulation of the interactions of plants with the environment using L-systems and their extensions.* PhD thesis, University of Calgary, October 1997.
28. R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. Proceedings of SIGGRAPH '96 (New Orleans, Louisiana, August 4–9, 1996) ACM SIGGRAPH, New York, 1996, pp. 397–410.
29. M. Nagl. On a generalization of Lindenmayer-systems to labelled graphs. In A. Lindenmayer and G. Rozenberg, editors, *Automata, languages, development*, pages 487–508. North-Holland, Amsterdam, 1976.
30. P. Prusinkiewicz. A look at the visual modeling of plants using L-systems. In R. Hofestädt, T. Lengauer, M. Löffler, and D. Schomburg, editors, *Bioinformatics*, Lecture Notes in Computer Science 1278, pages 11–29. Springer-Verlag, Berlin, 1997.
31. P. Prusinkiewicz. In search of the right abstraction: the synergy between art, science, and information technology in the modeling of natural phenomena. In C. Sommerer and L. Mignonneau, editors, *Art at Science*, pages 60–68. Springer, Wien, 1998.
32. P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Měch. Visual models of plant development. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages,* Vol. III*: Beyond words*, pages 535–597. Springer, Berlin, 1997.
33. P. Prusinkiewicz and J. Hanan. *Lindenmayer systems, fractals, and plants*, volume 79 of *Lecture Notes in Biomathematics.* Springer-Verlag, Berlin, 1989 (second printing 1992).
34. P. Prusinkiewicz and J. Hanan. L-systems: From formalism to programming languages. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 193–211. Springer-Verlag, Berlin, 1992.
35. P. Prusinkiewicz, M. James, and R. Měch. Synthetic topiary. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994). ACM SIGGRAPH, New York, 1994, pp. 351–358.
36. P. Prusinkiewicz and L. Kari. Subapical bracketed L-systems. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph grammars and their application to computer science; Fifth International Workshop*, Lecture Notes in Computer Science 1073, pages 550–564. Springer-Verlag, Berlin, 1996.
37. P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants.* Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.
38. P. M. Room, J. S. Hanan, and P. Prusinkiewicz. Virtual plants: new perspectives for ecologists, pathologists, and agricultural scientists. *Trends in Plant Science*, 1(1):33–38, 1996.
39. G. Rozenberg and A. Salomaa. *The mathematical theory of L systems.* Academic Press, New York, 1980.
40. A. R. Smith. Plants, fractals, and formal languages. Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 22–27, 1984). In *Computer Graphics*, 18, 3 (July 1984), pages 1–10, ACM SIGGRAPH, New York, 1984.
41. J. H. Woodger. *The axiomatic method in biology.* University Press, Cambridge, 1937. With appendices by A. Tarski and W. F. Floyd.

This article was processed using the LaTeX macro package with LLNCS style