Design and Implementation of the L+C Modeling Language

Radoslaw Karwowski and Przemyslaw Prusinkiewicz
Department of Computer Science, University of Calgary
Calgary, Canada T2N 1N4
Email: radekk|pwp@cpsc.ucalgary.ca

Abstract

L−systems are parallel grammars that provide a theoretical foundation for a class of programs used in the simulation of plant development and procedural image synthesis. In particular, the formalism of L−systems guides the construction of declarative languages for specifying input to these programs. We outline key factors that have motivated the development of L−system−based languages in the past, and introduce a new language, L+C, that addresses the shortcomings of its predecessors. We also describe a simulation program, lpfg, which makes it possible to execute models specified in L+C. To this end, L+C programs are translated into C++, compiled into a DLL, and linked with lpfg at runtime. The use of this strategy simplifies the implementation of the modeling system.

Reference

R. Karwowski, P. Prusinkiewicz: Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science* 86 (2), 19 pp.

Design and Implementation of the L+C Modeling Language

Radoslaw Karwowski [1] and Przemyslaw Prusinkiewicz [2]

Department of Computer Science
University of Calgary
Calgary, Canada

Abstract

L-systems are parallel grammars that provide a theoretical foundation for a class of programs used in the simulation of plant development and procedural image synthesis. In particular, the formalism of L-systems guides the construction of declarative languages for specifying input to these programs. We outline key factors that have motivated the development of L-system-based languages in the past, and introduce a new language, L+C, that addresses the shortcomings of its predecessors. We also describe a simulation program, lpfg, which makes it possible to execute models specified in L+C. To this end, L+C programs are translated into C++, compiled into a DLL, and linked with lpfg at runtime. The use of this strategy simplifies the implementation of the modeling system.

1 Background

L-systems were conceived as a rule-based formalism for reasoning on developing multicellular organisms that form linear or branching filaments [13]. Soon after their introduction, L-systems also began to be used as a foundation for visual modeling and simulation programs, and computer languages for specifying the models [3]. Subsequently, they found applications in the generation of fractals [16,28] and geometric modeling [26]. A common factor uniting these diverse applications is the treatment of structure and form as a result of development. A historical perspective of L-system-based software and its practical applications is presented in [17].

According to the L-system approach, a developing structure is represented by a string of symbols over an alphabet V. These symbols represent different

[1] E-mail: radekk@cpsc.ucalgary.ca
[2] E-mail: pwp@cpsc.ucalgary.ca

components of the structure, e.g., points and lines of a geometric figure, cells of a bacterium, or apices and internodes of a higher plant. The process of development is characterized in a declarative manner using a set of productions over V. During the simulation of development, these productions are applied in parallel steps to all symbols in the string, thus capturing the development in discrete time slices.

Lindenmayer [14] observed that L-system productions can be specified using standard notation of formal language theory. In the simplest, context-free case, productions thus have the form:

$$predecessor \longrightarrow successor,$$

where *predecessor* is a letter of alphabet V, and *successor* is a (possibly empty) word over V. For example, the division of a cell A into cells B and C can be written as $A \longrightarrow BC$. In the context-sensitive case, productions are often written as

$$lc < predecessor > rc \longrightarrow successor,$$

where symbols $<$ and $>$ separate the strict predecessor from the left context lc and the right context rc [19]. Both contexts are words over V. For example, the production pair:

$$\begin{aligned} Y < A > O &\longrightarrow LYS \\ O < A > Y &\longrightarrow SYL \end{aligned}$$

describes asymmetric division of a mother cell A into a short daughter cell S and long daughter cell L, separated by a cell wall Y. The sequence of these cells in the filament is guided by the state of the walls that delimit the mother cell, which may be young (Y) or old (O). Obviously, a complete description of the filament's development would also require productions that characterize the growth of cells and walls over time.

Early L-system-based programming languages closely followed the above notation [3,19]. However, the need to express increasingly complex models led to the addition of constructs found in other programming languages. A pivotal moment in this evolution was the introduction of parametric L-systems [9,20,24] and related formalisms (e.g., [4]), which associated numerical attributes to L-system symbols in a manner similar to attribute grammars [11]. This created a need for calculating parameter values in the production successor, given the values in the predecessor and its context. According to the original definition of parametric L-systems, these calculations were specified as arithmetic operations on the argument parameters, e.g.

$$\begin{aligned} A(x) &\longrightarrow A(2 * x)\ , \\ A(x) < B(y) > A(z) &\longrightarrow C(x + y)D(y + z)\ . \end{aligned}$$

For instance, an application of these productions to the predecessor string $A(1)B(2)A(3)$ would yield the successor string $A(2)C(3)D(5)A(6)$. L-system

symbols with the associated parameters have been termed *modules*[3].

In modeling practice, entire procedures soon became needed to calculate new parameter values. Recognizing this need, Hanan [9] introduced the following syntax for L-system productions:

$$lc < \text{predecessor} > rc\ \{\alpha\} : cond\ \{\beta\} \longrightarrow \text{successor}.$$

Here α and β are C-like compound statements, and *cond* is a logical expression that guards production application. A production is applied in stages. First, it is determined whether the production predecessor *pred*, surrounded by the left context *lc* and the right context *rc*, matches the given symbol in the string. If this is the case, the compound statement α is executed, and the condition *cond* is evaluated. If the result of this evaluation is non-zero ('true'), the second compound statement β is also executed. On this basis, parameters values in the production successor are determined, and the successor is inserted into the resulting string. For example, the following is a valid production:

$$A(x) < B(y) > C(z)\ \{r = x * x + y * y + z * z;\} : r > 2\ \{t = x + y + z;\} \longrightarrow D(t)E(2 * t).$$

At the top level, an L-system with productions in the above form operates in a declarative fashion, by rewriting elements of a string according to their type, context, and associated parameters. Within each production, however, calculations are performed sequentially, using constructs borrowed from an imperative language. This combination of paradigms suggests two alternative strategies for defining L-system-based languages [21]:

- extend the formal notation for productions with constructs borrowed from an imperative language, or
- extend an existing imperative language with constructs inherent in L-systems.

The modeling program cpfg [9] and its modeling language [22] are representative of the first approach. The interpreter of the cpfg language was constructed following the standard steps of lexical analysis and parsing of the input language. In spite of the well-developed methodology for translator construction (e.g. [2]), however, writing the interpreter or compiler of a comprehensive language is a large task. Consequently, the cpfg language only includes a limited subset of C-like statements; for example, it does not support user-definable functions and typed parameters associated with the modules. As a result, while simple L-system models can be expressed using the cpfg language in

[3] This use of the term *module* originates in biology, where repetitive components of plant architecture are commonly referred to as modules. In many applications, L-system symbols with associated parameters represent modules in the biological sense of the word, and therefore are denoted using the same term.

an elegant, compact manner, specification and maintenance of larger models becomes difficult.

In the second case, an existing "base" language is extended with support (e.g., classes, libraries, or new syntactic constructs) specific to L-systems. This makes it possible to take advantage of the programming tools developed for the base language while programming with L-systems. Pursuing this idea, Hammel [8] implemented differential L-systems [18] in *SIMULA*, and Erstad [5] implemented an environment for programming with L-systems in *LISP*. Both implementations preserve the syntax of their base languages. In contrast, the L+C language, which we describe in this paper, extends the syntax of C++. We describe here the design and implementation of L+C following its first definition in [23], and the subsequent refinement and implementation in [10].

2 The L+C modeling language

The key new conceptual elements introduced in L+C are:

- typed module parameters, including all primitive and compound data types (structures) supported by C++,
- productions with multiple successors,
- extension of the notion of context-sensitivity with 'new context' constructs, which speed up information transfer across simulated structures.

In addition, by virtue of being based on the C++ language, L+C has the full expressive power of C++.

At the top level, an L+C program is a set of declarations for:

- structures and classes,
- global variables,
- functions,
- the derivation length,
- modules
- the axiom,
- pattern matching options,
- productions,
- decomposition rules,
- interpretation rules,
- control statements.

The declarations of structures, classes, variables and functions have exactly the same syntax and meaning as in C++. The remaining declarations are specific to L+C, and are described below.

2.1 Derivation length specification

Derivation length is the number of derivation steps to be performed during the simulation. It is specified using the syntax:

`derivation length:` *integer_expression*;

The *integer_expression* is evaluated prior to the first derivation step.

2.2 Module declarations

In L+C, a module consists of an identifier (which must be a valid C++ identifier [27]) and an optional list of parameters. Modules must be declared before they are used. The declaration specifies the number and types of parameters that are associated with the given module type according to the following syntax:

`module` $identifier(parameter\_list_{opt})$;

Examples of valid module declarations are:

```
module A(); // module A with no parameters
module N(float); // module N with one parameter of type float
module Metamer(int, MetamerData); // module Metamer with
     // parameters of type int and (user-defined) MetamerData
```

2.3 Axiom declaration

The axiom declaration specifies the initial L-system string using the following syntax:

`axiom:` *parametric_string*;

where *parametric_string* is a non-empty string of modules. For instance, if the modules have been declared as in Section 2.2, and `s_init` is a structure of type `MetamerData`, the following is a valid axiom declaration:

```
axiom: Metamer(1, s_init) N(0.25) A();
```

2.4 Specification of productions

The syntax of productions is a combination of formal L-system notation and the C++ syntax for function definitions:

predecessor :
{
    production_body
}

The predecessor has one of the following forms:

$$new\_left\_context << left\_context < strict\_predecessor > right\_context$$
$$left\_context < strict\_predecessor > right\_context >> right\_new\_context$$

The strict predecessor specifies the part of the string being rewritten by the production. It can be a single module, as assumed in the usual definition of L-systems, or a string of several modules, as defined for pseudo-L-systems [16]. The optional left and right contexts are strings of modules that need to be in the neighborhood of the strict predecessor in order for the production to

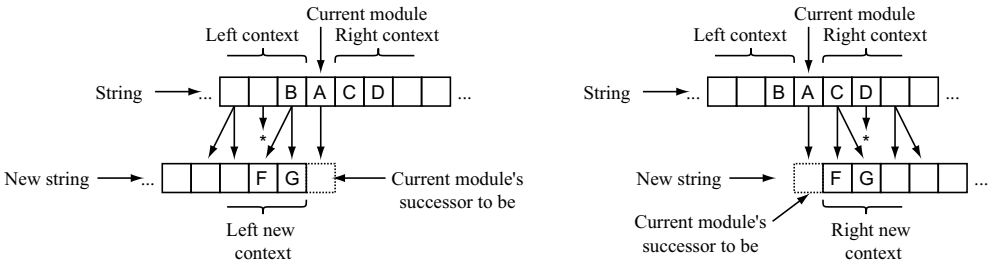

Fig. 1. The context of L-system productions. The left new context is available if the successor string is built from left to right (left figure). The right new context is available if the successor string is built from right to left (right figure).

apply. In contrast, the new contexts specify modules that must be present in the neighborhood of the production successor, i.e., in the string being derived. This information is easily available if the string is being rewritten in a particular direction: from left to right in the case of new left context, and from right to left in the case of new right context (Figure 1). L+C supports predefined functions `Forward()` and `Backward()`, usually called within control statements (Section 2.8), to specify the direction of individual derivation steps. In theory, two-sided new context could also be defined, but this is a much more involved concept and, therefore, it is not supported by L+C.

The parameters that appear in the production predecessor are formal parameters. All the formal parameters of every module in a production predecessor must be listed, even if they are not used in the production body. An example of a valid production predecessor that uses the modules declared in Section 2.2 is:

```
Metamer(il, dl) N(w) < Metamer(i, d) > A()
```

Formal parameters have types determined by the declarations of the modules. They are bound to the actual parameters in the string during production application [20,24]. The scope of the formal parameters is the same as the scope of formal parameters in C++ functions.

The production body is a compound statement that may contain any code allowed inside a C++ function. In addition, the production body may include one or more `produce` statements, which specify possible successors of the production. The `produce` statement has the syntax:

`produce` *parametric_string*$_{opt}$;

where *parametric_string* is defined as in the axiom (Section 2.3). Each `produce` statement is the production's exit point, analogous to `return` in a C++ function. Thus, if several `produce` statements are present in the production body, the first executed statement will terminate the production application. Typically, the choice of alternative successors is controlled by C++ conditional statements.

2.5 Declaration of pattern matching options

When matching context-sensitive production predecessors to a string, it is often convenient to only consider some module types. In L+C, modules relevant to pattern matching may be specified using two mutually exclusive declarations:

`consider:` *list_of_module_names*;
`ignore:` *list_of_module_names*;

In the first case, only the explicitly listed module types will be considered while context matching. In the second case, all modules will be considered except for those on the list. For example, the declaration:

```
ignore: N;
```

states that modules of type `N` will be ignored. If this is the case, a production with the predecessor `Metamer(i, d) < A()` will be applicable to the module `A()` in the axiom of Section 2.3, because module `N` separating `Metamer` from `A` will be ignored.

2.6 Decomposition rules

As defined by Lindenmayer [13], L-systems operate in discrete derivation steps. Each step consists of a (conceptually) parallel application of suitable productions to all symbols in the predecessor string. This parallelism is intended to capture progression of time by a given interval, the same for all components of the modeled structure. Thus, for example, the L-system production $A \longrightarrow BC$ expresses the idea "module A develops into modules B and C over a given time interval." In practice, it is often necessary to also express the idea that a given module is a compound module, consisting of several elements. A logical analysis of the notions "develops over time" and "consists of" was presented by Woodger [29]. Prusinkiewicz et al. [22,25] showed that, in a grammar setting, these notions correspond to L-system productions and Chomsky context-free productions, respectively.

In L+C, Chomsky productions are called decomposition rules. They are specified using the same syntax as context-free L-system productions, and are identified using the keyword `decomposition`, as in the following example:

```
decomposition:
Metamer(i, d) : { produce Internode(i, d) Leaf(d) Bud(); }
```

This production characterizes a `Metamer` as a compound module consisting of an `Internode`, a `Leaf`, and a `Bud`. We assume that these modules have been declared earlier in the L+C program.

The integration of decomposition rules into the L-system framework affects the way in which a derivation step is performed [22]. In L+C, decomposition rules are applied recursively, after the definition of the initial string by the

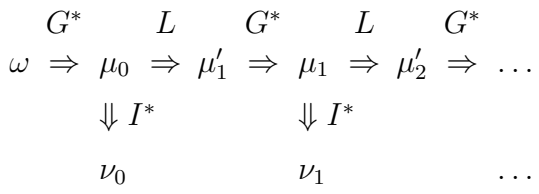

Fig. 2. Generation of a developmental sequence using an L-system with decomposition and interpretation rules. Beginning with the axiom ω, the progressions of strings μ_1, μ_2, μ_3, ... results from the interleaved application of decomposition rules G and L-system derivation steps L. The interpretation rules I map strings μ_i into strings ν_i. The strings ν_i are interpreted graphically.

`axiom` statement (Section 2.3) and after each step of standard L-system production applications (Section 2.4).

2.7 Interpretation rules

Structures generated with L-systems may be visualized by assigning a graphical interpretation to a predefined set of modules [16,26,28]. For example, in L+C, a predefined module `F(float)` draws a line of a given length in the current direction (as defined in turtle geometry [1]); `Line2D(point2D, point2D)` draws a line between two given points, and `SetColor(int)` assigns a color to geometric primitives.

From the user's perspective, it is often more convenient to express the model in terms of modules inherent in the modeling domain (e.g., apices, internodes, and leaves in the case of plant models) rather than in terms of modules with a geometric interpretation (e.g., points, lines, and polygons). In order to separate these conceptual and visual aspects of model specification, Kurth [12] introduced the notion of interpretation rules. Interpretation rules are similar to decomposition rules in that they are context-free Chomsky productions, and are applied recursively, after each derivation step (specifically, after the decomposition rules have been applied). In contrast to decomposition rules, however, interpretation rules do not affect the outcome of the following derivation steps. Instead, they are applied 'on the side', creating modules that are passed to the graphical part of the modeling program, and discarded once they have been interpreted (Figure 2).

In L+C, interpretation rules are identified using the keyword `interpretation`, as in the following example:

```
interpretation:
Internode(i, d) : { produce SetColor(1) F(d.length); }
```

The above production specifies that module `Internode` will be represented graphically as a straight line `F`, drawn using the color with index 1. The line length is specified by field `length` in data structure `d`.

2.8 Control statements

Control statements were introduced by Hanan [9] (see also [22]) to define procedures that are executed at specific points of an L-system-based derivation. In L+C, they are specified using the syntax:

`Start`|`StartEach`|`EndEach`|`End` :
{
    compound_statement
}

The control statements are executed as follows:

- `Start` is executed at the beginning of the program,
- `StartEach` is executed before every derivation step,
- `EndEach` is executed after every derivation step,
- `End` is executed after the last derivation step.

Any code that is allowed inside a C++ function can be specified as the *compound_statement*. Typical uses of the control statements include initializing global variables, opening and closing I/O streams, and reporting simulation statistics after each simulation step.

3 Example

A sample L+C program that generates a branching structure is presented below:

```
1    #include <lpfgall.h>
2    #include <math.h>
3
4    const int Delay = 1;
5    const float BranchingAngle = 45.0;
6    const float LengthGrowthRate = 1.33;
7
8    derivation length:  17;
9
10   struct InternodeData { float length, area; };
11
12   module Apex(int,float);
13   module Metamer(float);
14   module Internode(InternodeData);
15
16   Start:  { Backward(); }
17   ignore:  Right;
```

```
18
19  axiom:  Apex(0,BranchingAngle);
20
21  Apex(t,angle) :
22  {
23       if (t<0) // young apex
24            produce Apex(t+1,angle);
25       else // mature apex
26            produce Metamer(angle) Apex(0,-angle);
27  }
28
29  Internode(id) >> SB() Internode(id2) EB() Internode(id3) :
30  {
31       id.area = id2.area + id3.area;
32       id.length *= LengthGrowthRate;
33       produce Internode(id);
34  }
35
36  Internode(id) >> Internode(idr) :
37  {
38       id.area = idr.area;
39       id.length *= LengthGrowthRate;
40       produce Internode(id);
41  }
42
43  Internode(id) >> Apex(t,angle):
44  {
45       id.length *= LengthGrowthRate;
46       produce Internode(id);
47  }
48
49  decomposition:
50  Metamer(angle) :
51  {
52       InternodeData id = {1, 1};
53       produce
54            Internode(id)
55            SB() Right(angle) Apex(-Delay,angle) EB()
56            Internode(id);
57  }
58
59  interpretation:
```

```
60   Internode(id) :
61   {
62        produce SetColor(2)
63             SetWidth(pow(id.area,.5)) F(id.length);
64   }
```

The modeled structure consists of three types of modules, which are given biologically meaningful names `Apex`, `Metamer`, and `Internode` (lines 12-14). The process of string derivation is performed from right to left, as indicated by calling the reserved L+C function `Backward()` in the `Start` statement (line 16). In the process of context matching, module `Right` (used to specify the branching angle in line 55) is ignored (line 17). The initial structure defined by the axiom is a single apex (line 19). Its parameters characterize the developmental stage and the branching angle of the first branch that will be produced by this apex. According to the first production (lines 21-27), an immature apex will grow older, and a mature apex will produce a metamer, over the time interval associated with a derivation step. The decomposition rule (lines 50-57) specifies that the metamer consists of two internode segments and a lateral branch delimited by the modules `SB()` (start branch) and `EB()` (end branch), which are predefined in L+C. The branch initially consists of a lateral apex, placed at a given angle with respect to its supporting internode. The development of internodes is described by the three productions in lines 29 to 47. These productions specify that an internode will grow in length by factor `LengthGrowthRate` (line 6) in each derivation step. They also determine the cross-section area of each internode as the sum of the cross-sections of the

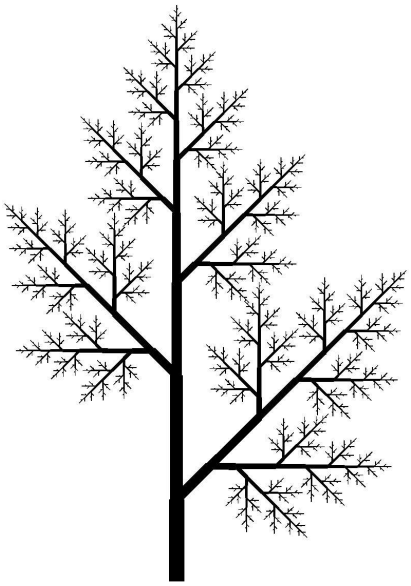

Fig. 3. Example of a structure generated by the sample L-system

internodes supported by it. Specifically, the *new context* construct is used to accumulate the cross-section of branches when moving from the apices toward the base of the structure. Finally, the interpretation rule (lines 60-64) specifies that each internode will be visualized as a line of length and width determined by the internode parameters. The structure generated by this L-system is shown in Figure 3.

4 Translation and execution of L+C programs

To execute models specified in the L+C language, such as the model discussed above, we have created the simulation program lpfg. From a user's perspective, lpfg accepts L+C models as input, performs L-system derivations, and produces output in the form of images, developmental animations, and statistical data. Internally, however, we adopted a different strategy: we compile L+C models into a dynamically linked library module (DLL), and link it at run time with lpfg (Figure 4). As discussed in Section 1, this strategy greatly simplifies the task of compiling L+C programs.

In designing lpfg, we had to decide which aspects of the simulation would be executed by the DLL, and what would be delegated to lpfg. We based our decisions on the assumption that users would want to interactively experiment with the models, and therefore should experience as short a delay as possible between model submission and execution. To achieve this goal, we maximized the functionality of the simulator, and minimized the amount of executable code produced while compiling specific L+C models. In consequence, the simulator performs all of the generic L-system operations: traversing a string

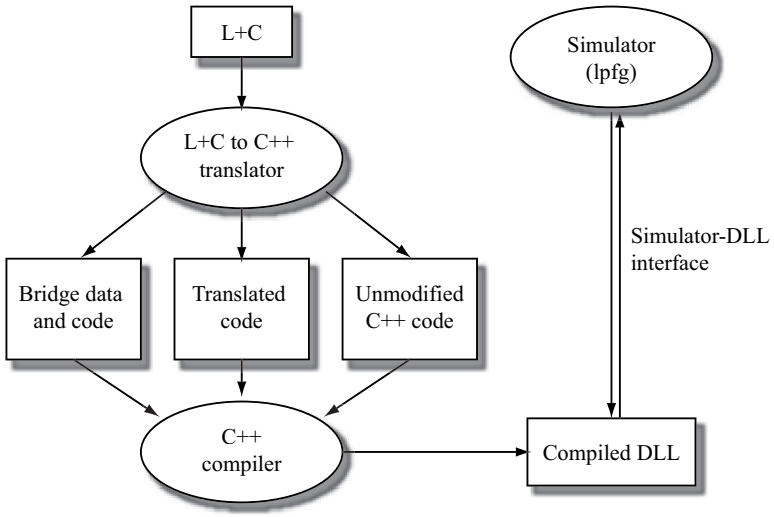

Fig. 4. Components of the modeling system. Fixed components are shown as ellipses, model-dependent components are shown as rectangles.

of modules, matching production predecessors to it, appending production successors to the successor string, and interpreting the string graphically. In general terms, these are operations that treat L-system strings as a topological space (c.f. [6,7]).

To compile L+C, we first translate the L+C source into C++ code, then compile it into a DLL using a standard C++ compiler. By pursuing this approach, we reduced the task of developing the L+C compiler to that of developing a translator from L+C to C++: a relatively minor task, given that most of the L+C syntax is also C++ syntax.

Translating L+C code to C++ yields three types of code (Figure 4):

- L+C-specific constructs are translated into C++,
- constructs that are already in C++ remain intact,
- additional *bridge code* is generated to interface lpfg and the model-dependent code.

The simulator performs an L-system string derivation given only the information that can be provided by the DLL at run time (since the simulator is a fixed component and is not recompiled for every L+C program). The bridge code is therefore needed to address problems caused by:

- the polymorphic nature of L-system strings, which consist of modules of arbitrary, user-defined types, and
- the unrestricted format of productions, which operate on modules of arbitrary types, in arbitrary contexts, and produce arbitrary sub-strings of modules as a result.

To illustrate the translation process, let us consider string derivation by lpfg in more detail. At the top level, string derivation is performed by the lpfg `Execute()` function, defined below:

```
void Execute()
{
     Start();
     Axiom();
     DecomposeString();
     for (int i=0; i<DerivationLength(); ++i)
     {
          StartEach();
          Derive();
          DecomposeString();
          EndEach();
     }
     End();
}
```

The underlined functions are defined in the process of translating the L+C program to C++ to perform the following tasks:

- `Start()`, `StartEach()`, `EndEach()`, and `End()` execute the compound statements specified in the corresponding L+C control statements (Section 2.8);
- `Axiom()` creates the initial L-system string (Section 2.3),
- `DerivationLength()` returns the value specified in the L-system `derivation length` statement (Section 2.1).

The generation of these functions from the L+C source is straightforward (except for the `Axiom()` function, which is handled similarly to the `produce` statement discussed later on). For example, the translation of the L+C `Start` statement is given below:

Original code	Translated code

`Start:`
{
...
}
	`void Start()`
	{
	...
	}

The remaining components of the `Execute()` function are the lpfg `Derive()` and `DecomposeString()` functions. They scan the predecessor string and create the successor string by calling L-system productions and decomposition rules (we refer to them jointly as "productions" from now on). In order to describe the translation of productions, let us consider the following sample L+C code (elements specific to L+C are underlined):

```
module A(data, float);
module B(int, float);

A(dl, xl) < B(n, a) :
{
     if (a>xl)
          produce B(n+1, xl);
     else
          produce B(n-1, xl);
}
```

The translation process is based on the fact that productions are similar to functions in imperative programming languages [21]. These similarities can be summarized as follows:

- A production is a piece of code to be executed;
- Its input is its predecessor and, optionally, the parameters of the predecessor's modules; and
- Its output is the successor.

On the other hand, productions and functions differ in two respects:

- L+C programs do not call productions explicitly. The mechanism of matching productions to string elements determines which production will be applied and when.
- Productions do not return a value in the traditional sense. Instead, their output is appended to the L-system successor string.

Given these similarities and differences, L+C productions are translated into C++ functions in a process that distinguishes three types of production components:

- The statements that constitute valid C++ code are copied verbatim into the C++ function body.
- The predecessor is translated into a function prototype, with an automatically generated name. Argument types are derived from the declaration of the relevant modules. For example, the following substitution is made:

Original code:	Translated code:
`A(dl, xl) < B(n, a)`	`void P1(data dl, float xl, int n, float a)`

- The `produce` statements are translated into blocks within the production body. The resulting code adds a successor to the new string and terminates production execution. In our example, the first `produce` statement is translated as follows:

Original code:	Translated code:
`produce B(n+1, xl);`	`{App(B_id); App(n+1); App(xl); return;}`

As mentioned before, the tasks of traversing the predecessor string and calling productions are performed by the lpfg functions `Derive()` (for regular productions) and `DecomposeString()` (for decomposition rules). To perform these tasks in a generic manner, independent of module types, lpfg uses low-level, untyped, internal representations of strings and production prototypes. Each module in the string is represented in a uniform manner by a module identifier, module size information, and a sequence of bytes that represent parameter values. The bridge code includes a data structure that specifies the types of modules in the strict predecessor and the context of each production.

Given these representations, lpfg can match productions to the string in a generic manner.

The bridge code also includes a set of *caller functions*, one for each production, which re-assemble the parameters needed to apply specific productions (e.g., arguments of function `P1` in the above example of predecessor translation) and call C++ functions representing the translated productions. The caller functions have a fixed prototype, and thus can be called by lpfg irrespective of the prototypes of productions with which they interface.

An inverse process takes place while appending production successors to the generated string. In this case, it is necessary to convert the typed representation of modules in the production bodies to their untyped representations in the successor string. This conversion is implemented by the C++ function template `App()`, which is instantiated by the C++ compiler for every returned module type.

Reviewing the production application from a methodological point of view [15], we combine a *homogeneous* handling of data polymorphism while traversing the string (the same code is used by lpfg to scan modules and perform pattern matching independent of module type) with a *heterogeneous* method for calling productions and returning results (separate functions are generated to call different productions, and to append modules of different types). This instancing is implemented in part by the L+C translator, which generates the caller functions, and in part by the C++ compiler, which instantiates the `App()` template. Technical details are given in [10].

5 Conclusions

We have described the modeling language L+C, which incorporates C++ into the framework of L-systems. We have also implemented a simulation program lpfg, which provides the run-time environment for L+C. To implement the L+C translator, we have separated constructs specific to L-systems from the constructs inherited from C++. The L-system-specific code is translated into C++ and combined with the C++ code taken verbatim from the L+C programs. C++ bridge code is also generated to assist in interfacing L+C programs with lpfg. The entire C++ code is translated into a DLL module using a standard C++ compiler, and linked at runtime with (precompiled) lpfg. During simulations, lpfg performs generic L-system operations such as string traversal, while the DLL executes specific productions. Due to this partitioning of tasks, DLL modules are typically small compared to lpfg. As a result, L+C programs compile and link quickly, in the order of one second on current Windows and Linux workstations. This allows for interactive manipulation and modification of models. The increased expressiveness of L+C, compared to the previous L-system based languages, makes it possible to create models

of greater complexity. L+C is currently being used to model aspects of plant genetics, physiology, and biomechanics.

Acknowledgments

The original specification of the L+C language (then named L) was developed jointly with Jari Perttunen and Risto Sievänen during their visit to Calgary in February, 1999 [23]. We hereby gratefully acknowledge their contribution to this work. We thank Jean-Louis Giavitto and Jurgen Vinju for discussions and pointers to the literature, and Pavol Federl, Peter MacMurchy, Lynn Mercer, and Colin Smith for insightful comments on earlier versions of this paper. The support of the Human Frontier Science Program, Natural Sciences and Engineering Research Council of Canada, and the L-studio user community is also gratefully acknowledged.

References

[1] H. Abelson and A. A. diSessa. *Turtle geometry.* M.I.T. Press, Cambridge, 1982.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools.* Prentice Hall, Englewood Cliffs, 1986.

[3] 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.

[4] T. W. Chien and H. Jürgensen. Parameterized L systems for modelling: Potential and limitations. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 213–229. Springer-Verlag, Berlin, 1992.

[5] K. A. Erstad. L-systems, twining plants, Lisp. Master's thesis, University of Bergen, January 2002.

[6] J.-L. Giavitto and O. Michel. MGS: A programming language for the transformation of topological collections. Research Report 61-2001, CNRS - Université d'Evry Val d'Esonne, Evry, France, 2001.

[7] J.-L. Giavitto and O. Michel. Data structures as topological spaces. In *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, volume 2509 of *Lecture Notes in Computer Science*, pages 137–150, 2002.

[8] M. Hammel. *Differential L-systems and their application to the simulation and visualization of plant development.* PhD thesis, University of Calgary, June 1996.

[9] J. S. Hanan. *Parametric L-systems and their application to the modelling and visualization of plants*. PhD thesis, University of Regina, June 1992.

[10] R. Karwowski. *Improving the process of plant modeling: The L+C modeling language*. PhD thesis, University of Calgary, October 2002.

[11] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):191–220, 1968.

[12] 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.

[13] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[14] A. Lindenmayer. Developmental systems without cellular interaction, their languages and grammars. *Journal of Theoretical Biology*, 30:455–484, 1971.

[15] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languges, Paris, France, January 1997*. ACM, 1997.

[16] P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphics Interface '86 — Vision Interface '86*, pages 247–253, 1986.

[17] 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.

[18] P. Prusinkiewicz, M. Hammel, and E. Mjolsness. Animation of plant development. Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993). ACM SIGGRAPH, New York, 1993, pp. 351–360.

[19] P. Prusinkiewicz and J. Hanan. *Lindenmayer systems, fractals, and plants*, volume 79 of *Lecture Notes in Biomathematics*. Springer-Verlag, Berlin, 1989.

[20] P. Prusinkiewicz and J. Hanan. Visualization of botanical structures and processes using parametric L-systems. In D. Thalmann, editor, *Scientific visualization and graphics simulation*, pages 183–201. J. Wiley & Sons, Chichester, 1990.

[21] 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.

[22] P. Prusinkiewicz, J. Hanan, and R. Měch. An L-system-based plant modeling language. In M. Nagl, A. Schürr, and M. Münch, editors, *Applications of graph transformations with industrial relevance*, Lecture Notes in Computer Science 1779, pages 395–410. Springer-Verlag, Berlin, 2000.

[23] P. Prusinkiewicz, R. Karwowski, J. Perttunen, and R. Sievänen. Specification of L — a plant modeling language based on L-systems. Version 0.5. Internal report, Department of Computer Science, University of Calgary, 1999.

[24] 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.

[25] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The use of positional information in the modeling of plants. Proceedings of SIGGRAPH 2001 (Los Angeles, California, August 12–17, 2001). ACM SIGGRAPH, New York, 2001, pp. 289–300.

[26] P. Prusinkiewicz, F. Samavati, C. Smith, and R. Karwowski. L-system description of subdivision curves. To appear in the *International Journal of Shape Modeling*, 2003.

[27] B. Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, 1987.

[28] A. L. Szilard and R. E. Quinton. An interpretation for DOL systems by computer graphics. *The Science Terrapin*, 4:8–13, 1979.

[29] J. H. Woodger. *The axiomatic method in biology*. University Press, Cambridge, 1937. With appendices by A. Tarski and W. F. Floyd.