doi:10.1093/aob/mcr121, available online at www.aob.oxfordjournals.org

ANNALS OF BOTANY Founded 1887

PART OF A SPECIAL ISSUE ON FUNCTIONAL–STRUCTURAL PLANT MODELLING

Towards aspect-oriented functional–structural plant modelling

Mikolaj Cieslak[1,2,†], Alla N. Seleznyova[2], Przemyslaw Prusinkiewicz[3] and Jim Hanan[4,*]

[1]*The University of Queensland, School of Mathematics and Physics, Qld 4072, Australia,* [2]*The New Zealand Institute for Plant & Food Research Limited, Palmerston North 4442, New Zealand,* [3]*Department of Computer Science, University of Calgary, AB T2N 1N4, Canada and* [4]*The University of Queensland, Centre for Biological Information Technology, Qld 4072, Australia*

** For correspondence. E-mail j.hanan@uq.edu.au*

[†]*Present address: INRA, Virtual Plants INRIA Team, UMR AGAP, TA A-108/02, 34398 Montpellier Cedex 5, France.*

Received: 3 January 2011 Returned for revision: 4 February 2011 Accepted: 7 March 2011

• *Background and Aims* Functional–structural plant models (FSPMs) are used to integrate knowledge and test hypotheses of plant behaviour, and to aid in the development of decision support systems. A significant amount of effort is being put into providing a sound methodology for building them. Standard techniques, such as procedural or object-oriented programming, are not suited for clearly separating aspects of plant function that criss-cross between different components of plant structure, which makes it difficult to reuse and share their implementations. The aim of this paper is to present an aspect-oriented programming approach that helps to overcome this difficulty.

• *Methods* The L-system-based plant modelling language L+C was used to develop an aspect-oriented approach to plant modelling based on multi-modules. Each element of the plant structure was represented by a sequence of L-system modules (rather than a single module), with each module representing an aspect of the element's function. Separate sets of productions were used for modelling each aspect, with context-sensitive rules facilitated by local lists of modules to consider/ignore. Aspect weaving or communication between aspects was made possible through the use of pseudo-L-systems, where the strict-predecessor of a production rule was specified as a multi-module.

• *Key Results* The new approach was used to integrate previously modelled aspects of carbon dynamics, apical dominance and biomechanics with a model of a developing kiwifruit shoot. These aspects were specified independently and their implementation was based on source code provided by the original authors without major changes.

• *Conclusions* This new aspect-oriented approach to plant modelling is well suited for studying complex phenomena in plant science, because it can be used to integrate separate models of individual aspects of plant development and function, both previously constructed and new, into clearly organized, comprehensive FSPMs. In a future work, this approach could be further extended into an aspect-oriented programming language for FSPMs.

Key words: L-system, aspect-oriented programming, *Actinidia deliciosa* (kiwifruit), functional–structural plant model, plant architecture, carbon dynamics, biomechanics, hormone transport.

INTRODUCTION

Functional–structural plant models (FSPMs) have attracted significant interest and the expertise of researchers in botanical, mathematical and computational sciences. The motivations for developing these models range from increasing our fundamental understanding of plant behaviour to developing decision support systems that will aid in optimizing horticultural, agricultural and forestry production (Vos *et al.*, 2010). The diverse applications of FSPMs create a growing need to share and reuse not only complete models, but also model components and software tools (cf. Pradal *et al.*, 2008). In this context, an important open problem is to develop methods for integrating previously modelled aspects of plant function into a single model (Prusinkiewicz, 2009). Unlike the decomposition of a model into components (modules) such as internodes, buds, leaves, flowers, fruits and root structures, aspects represent decomposition into functions such as long-distance signalling and growth regulation by hormones, photosynthesis, transport and allocation of water, nutrients and sugars, and the response of plants to gravity. Here we describe an L-system-based technique for constructing FSPMs from models of individual aspects that can easily be shared, reused and recombined. This ability to model and combine individual aspects is essential to the further progress of FSPMs.

From a programming perspective, the key difficulty in decomposing plant models into both components (modules) and functions (aspects of model operation) is that they represent criss-crossing concerns: different modules may exhibit the same function, and different functions may be shared by the same module. For example, the progress of time (ageing) is an aspect that affects most structural modules, while these modules are also likely to be endowed with other functions. A straightforward implementation of the ageing process would require a duplication of the corresponding code in all modules. Here we introduce a technique for constructing FSPMs that removes this duplication by encapsulating, making reusable and integrating the code that expresses each concern. Our approach is inspired by the aspect-oriented programming paradigm introduced by Kiczales *et al.* (1997), but our technique is specific to FSPMs expressed using L-systems.

In principle, a general-purpose programming language, such as C++, can be used to model plant structure and function, but the implementation details tend to hide the essence of the models and make them inaccessible and incomprehensible for non-expert programmers (Prusinkiewicz, 1998). In particular, general-purpose programming languages do not inherently capture the topological connections between components of a developing plant's branching structure, and thus individual models must incorporate the code for representing and manipulating these connections (Prusinkiewicz, 2009). As an alternative, L-systems (Lindenmayer, 1968, 1971) have been used as the basis for a special-purpose language for modelling plants (Prusinkiewicz *et al.*, 2000*a*). They are well suited to describing growing linear or branching structures, and are the most widely used plant-modelling formalism (Fourcaud *et al.*, 2008; Vos *et al.*, 2010). In particular, parametric L-systems (Prusinkiewicz and Lindenmayer, 1990; Hanan, 1992) can capture the complexity of plant architecture and physiological processes within a single model by providing a means of describing activities of individual plant elements. This is achieved through a set of rules that are applied to all instances of a particular element, irrespective of the number of its occurrences within the structure, in each of a series of time steps. Many existing L-system models integrate the development of a plant structure with some aspects of its function, such as long-distance signalling and apical dominance (Prusinkiewicz *et al.*, 2009), metabolic regulatory pathways and genetic processes (Buck-Sorlin *et al.*, 2005), acquisition, transport and allocation of carbon (Allen *et al.*, 2005; Lopez *et al.*, 2008; Cieslak *et al.*, 2011) or dry matter (Fournier and Andrieu, 1999), nitrogen distribution (Bertheloot *et al.*, 2008), metabolic processes regulating growth (Perttunen and Sievänen, 2005), bending of branches due to external forces (Jirasek *et al.*, 2000; Taylor-Hell, 2005; Costes *et al.*, 2008), auto-regulation of nodulation (Han *et al.*, 2010), and responses to pathogens or insects (Hanan *et al.*, 2002). How can these previously modelled aspects be combined into a single, integrative L-system model?

There are several L-systems-based programming languages that could be used to illustrate our technique. Here we focus on the L+C language (Karwowski and Prusinkiewicz, 2003; Prusinkiewicz *et al.*, 2007*b*) because of its advanced features and existing collection of L-system models, but the technique is applicable to other L-system-based languages as well, for example L (Perttunen and Sievänen, 2005) or XL (Hemmerling *et al.*, 2008). After an overview of the relevant features of L+C, we present a programming technique that allows for the easy integration of previously modelled aspects. We illustrate it with a model of a kiwifruit shoot that integrates previously modelled aspects of the shoot's architectural development, carbon allocation, growth regulation by auxin, and biomechanics.

The L+C language

Karwowski and Prusinkiewicz (2003) presented all of the key features of the L+C modelling language, which were updated by Prusinkiewicz *et al.* (2007*b*). Among other things, L+C allows for typed module parameters (primitive and compound data types), function calls, productions with multiple successors (e.g. through the use of conditional statements in the productions) and, most significantly, fast information transfer. These features make it more flexible than its predecessor, the *cpfg* language.

In an L-system, plant components are represented by modules, which together form a string representing the plant. In L+C, a module is declared using the keyword 'module' followed by the name of the module (e.g. a letter) (Karwowski and Prusinkiewicz, 2003) and may have several parameters, with the parameter types specified within the parentheses immediately following the declaration of a module. For example,

```
struct data { float x, y; };
module Z(data, int);
```

where `data` is a compound data type consisting of two floating-point variables (real numbers), and the module `Z` has two parameters: one is of type `data` and the other is `int` (integer).

The initial L-system string is declared after the keyword `axiom`. For each derivation step in a simulation, rewriting rules specify how a predecessor module is replaced by any number of successor modules. For a given time step *t*, several derivation steps may be performed. The total number of derivation steps is specified using the command `derivation length: N;`, where `N` is an integer greater than zero. Although, according to the original definition of L-systems, all the modules are rewritten in parallel, in practice, the string is rewritten one module at a time in either left-to-right or right-to-left order.

A rewriting rule is specified by a predecessor, representing the component to be replaced in the string, followed by a colon, and delimited by curly braces, with any number of `produce` statements denoting the possible successors to replace that predecessor. Also, several rewriting rules may be arranged into subsets using the keyword `group`, and one of these subsets may be invoked using the predefined function `UseGroup(n)`, where `n` is the group number. In that case, only the rewriting rules within group *n* are applied to the string, for instance:

```
StartEach:{UseGroup(1);}
EndEach:{}
// module declaration, as above
derivation length: 10;
data dataInit={1.0, 2.0};
axiom: Z(dataInit,4);
group 1:
Z(a,b): {
  ...update a.x, a.y, and b
  if (b>0)
    produce Z(a,b);
  else
    produce;
}
```

where `StartEach`/`EndEach` are predefined control statements that are executed at the beginning/end of each derivation step. The flexibility of L+C is that the rewriting rules may contain any valid C++ constructs, such as conditional statements and function calls. Furthermore, the rewriting rules may be context-sensitive, where the module and its neighbouring modules are matched to the predecessor of the rule. For example, the following rule

```
Z(aL,bL)<Z(a,b)>Z(aR,bR):
```

is applied only when there is a left and right neighbour that match the modules to the left and right of the $<$ and $>$ symbols denoting left- and right-context, respectively. Either context may be left out if not required. Some modules may be ignored or considered in the context by specifying a list of modules at the start of the L-system using the keyword `Ignore` or `Consider`, respectively. These types of context-sensitive rules, combined with all the modules' parameters, are useful for modelling information transfer between neighbouring modules in a developing structure. The only drawback is that it takes $N - 1$ derivation steps to propagate information from one end to another in an N-element string (Prusinkiewicz *et al.*, 2007*b*). To accelerate the flow of information in one direction through the string, L+C includes a construct for fast information transfer, where a module can refer to the state of one of its newly produced neighbours in the current derivation step. This is only possible because the string is, in practice, rewritten sequentially (instead of in parallel) during a derivation step, and therefore to use fast information transfer, the direction of the derivation must be set. For instance, assuming acropetal flow up the plant, in the rule

```
Z(aL,bL) << Z(a,b):
```

the symbols $\ll$ indicate that the rule considers values of the parameters in the new left context module `Z(aL,bL)`, which have already been updated in the current derivation step, instead of being from the previous derivation step as is the case in regular context-sensitive rules. A similar rule can be written for basipetal flow as in

```
Z(a,b) >> Z(aR,bR):
```

where the parameters of the new right context `Z(aR,bR)`, indicated by the symbols $\gg$, have already been updated in this step. In L+C, the direction of flow is set using the predefined function `Forward()` for left-to-right derivation (acropetal), and `Backward()` for right-to-left derivation (basipetal). This advancement in L+C provides a significant speed-up for simulating propagation of hormones, resources and mechanical forces through a plant; however, it is difficult to integrate more than one such aspect of plant function into a single model (Prusinkiewicz *et al.*, 2007*b*). The difficulty is in building well-structured models, with each aspect implemented and tested independently, and, then, easily combining them into one model. Now, let us examine the proposed programming technique that makes this possible in L+C.

METHODS: A MULTI-MODULE APPROACH

In the L-system-based language L+C, there are two common ways to represent components of the developing plant structure using modules. The first is the most standard way, where a separate module is used for each component: `Internode()`, `Leaf()`, `Fruit()`, etc. This approach has the disadvantage that aspects shared among all modules must be specified separately for each module, even if the implementation is exactly the same. For example, the code for increasing the age of an organ must be repeated in several rules as follows:

```
Internode(age):{produce Internode(age+DT);}
Leaf(age): { produce Leaf(age+DT); }
...etc.
```

where `age` is the chronological age of the organ and `DT` is a user-defined time step. However, another approach can be used, where a single module represents all the different types of components, with one of its parameters defining the type, e.g. an `X(i)` module with parameter `i` giving the type: internode, leaf, etc. The advantage of this approach is that computations common to all modules can be specified with only one rule. For example, increasing the age of an organ can be accomplished in one rule as follows:

```
X(i,age): { produce X(i,age+DT); }
```

where `age` and `DT` are defined as before. Although this example is trivial, some computations carried out by productions can be much more complex, such as advanced numerical integration techniques. The disadvantage of this approach is that component-specific rule processing must be done manually using a conditional statement within a rewriting rule. For example, the above rule would change to

```
X(i,age): {
  switch(i) {
    case(internode):
      ...perform internode specific tasks
    case(leaf)
      ...perform leaf specific tasks
    case(...etc.
  }
  produce X(i,age+DT);
}
```

where the types of components can be defined using an enumeration in C++. This approach is, potentially, very tedious depending on the number of component types. It can lead to cryptic code, and seems to be contrary to the natural operation of the L-system formalism, where different modules are used to represent different components and the L-system simulator performs the rule matching automatically. Clearly, the advantages and disadvantages of these two approaches are the exact opposites of one another, i.e. no generic rules but automatic component-based rule matching or generic rules but manual component-based rule matching. Therefore, an L+C programming technique was developed to take advantage of both approaches.

To integrate different aspects of plant function in a single model, the proposed technique is to use pseudo-L-systems or pL-systems (Prusinkiewicz, 1986) combined with sets of productions (groups in L+C) that only consider modules specific to an aspect. In a pL-system, the number of modules in the strict predecessor of a rewriting rule may vary, as opposed to standard L-systems where only one module is allowed. This flexibility makes it possible to define components of a developing plant structure as a collection of modules, together forming one multi-module. Different modules in such a collection may represent different aspects of the same component of a plant, and rewriting rules can then be defined to apply only to those modules, i.e. an L+C group is defined for each aspect. Rewriting rules for these multi-modules can be used to facilitate communication between aspects, so that, for instance, the physical properties of an organ can be updated based on its associated aspects. In aspect-oriented programming terms, this is called weaving and represents the mechanism for coordinating aspects in our approach, so that models of individual aspects can be composed into one. For example,

```
#define GROW 1
#define ADVANCE_AGE 2
#define DT 0.1
int whichGroup=GROW;
StartEach: { UseGroup(whichGroup); }
EndEach: {
  if (whichGroup == GROW)
    whichGroup=ADVANCE_AGE;
  else whichgroup=GROW;
}
module Internode(float);
module Leaf(float);
module X(float);
derivation length: 20;
axiom: Internode(0) X(0.1) Leaf(0) X(0.1);
group GROW:
  Internode(length) X(age): {
    produce Internode(1 / (1+exp(-age+5)))
      X(age);
  }
  Leaf(area) X(age): {
    produce Leaf(1 / (1+exp(-age+5))
      X(age);
  }
group ADVANCE_AGE:
  X(age): { produce X(age+DT); }
```

where `Internode() X()` and `Leaf() X()` are multi-modules representing an internode and a leaf, respectively. In group `ADVANCE_AGE`, the task of increasing the age of each organ (regardless of type) is performed by a rule using the module `X()`. In group `GROW`, the length of an internode and the area of a leaf are updated using a simple logistic function based on age. In this case, the `Internode()` and `Leaf()` modules can be considered as the defining module of a multi-module, because they are used to distinguish one collection of modules from another.

The rewriting rules in the example given above are context free, but in the general case of context-sensitive productions, an additional construct is needed to consider only the context modules that are relevant to the aspect under consideration. For example, examine the rule

```
X(age,s) >> X(ageR,sR): {
  produce X(age+DT,sR);
}
```

in which an extended `X()` module is used to propagate signal `s` from right to left. In principle, the predecessor of this rule would not be matched if the `X()` modules in the rewritten string were separated by other modules, such as `Internode()` and `Leaf()`. L+C makes it possible to disregard these intervening modules using the `Ignore: Internode() Leaf();` or `Consider: X();` statements, which ensure that only `X()` modules will be considered as context. According to the original definition of L+C, the list of ignored or considered modules is global within the entire L-system model. To allow for a more flexible context specification, we extended L+C and *lpfg* with constructs for defining a separate list of modules to be considered or ignored within each group. These group-limited lists are expressed in the same way as the global list, using the keywords `Consider:` or `Ignore:` followed by a list of modules, but are written after the `group` statements rather than outside of the production set and have precedence over the global list of modules.

This approach, using a collection of modules to represent components of the plant, has both advantages over the two currently available approaches in L+C: generic rewriting rules for particular aspects and automatic component-based rule matching. Figure 1 shows a schematic of how multi-modules relate to the current approach used in L+C. As the examples given in this section are very simple, let us now consider modelling a kiwifruit shoot that integrates several non-trivial aspects.

RESULTS: APPLICATION TO INTEGRATING SEVERAL ASPECTS IN A SINGLE KIWIFRUIT SHOOT MODEL

To demonstrate the new approach, we present a kiwifruit shoot model that integrates architectural development with carbon dynamics, apical dominance and biomechanics, using the multi-module approach. The model is based on existing models of individual aspects, so the aim here is to show how these existing models can be integrated into one without major changes. The comprehensive kiwifruit shoot model is available on the L-studio/VLAB website: www.algorithmicbotany.org.

Architectural aspect

A kiwifruit shoot develops from a mature first-order axillary bud on a parent cane, i.e. the current season's shoot originates from an axillary meristem from the previous season (Foster *et al.*, 2007). The axillary bud on this cane initiates a number of preformed metamers before winter dormancy. In spring, after budbreak, there are two possible fates for the shoot: (1) terminate growth before most of these preformed metamers expand, or (2) continue growth until the end of the season, with all preformed metamers expanding and the shoot apical meristem initiating new metamers. The rate of metamer appearance corresponds to the time interval between successive appearance of leaves (phyllochron), which is modulated by temperature and other factors.

For the kiwifruit vine, Cieslak *et al.* (2011) modelled shoot development as a stochastic process, where the physical states of a shoot apex were represented by states in a discrete-time Markov chain. The process was then simulated using the

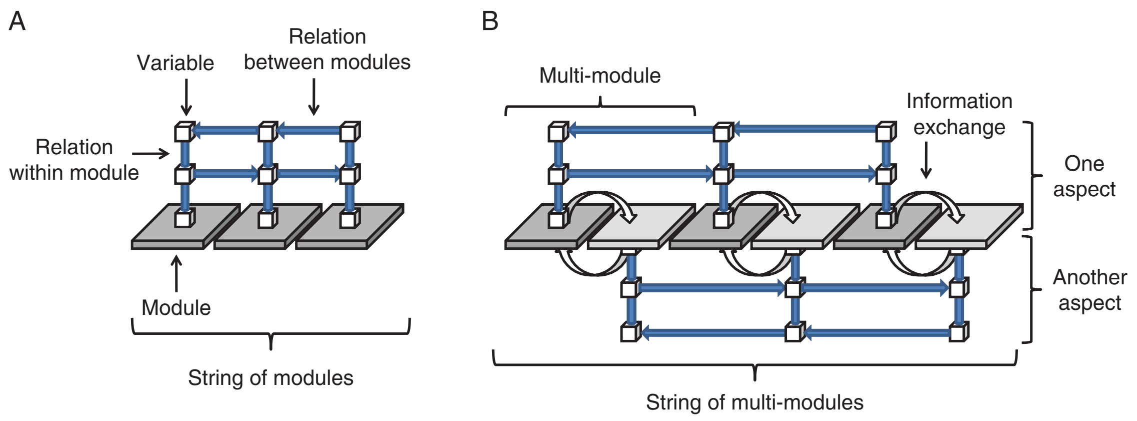

Fig. 1. A schematic representation of the multi-module approach. In an L-system, a structure consists of a string of modules, with each one representing a component of the structure. A module's variables are related through equations within the module itself and between neighbouring modules through context (A, redrawn from Prusinkiewicz, 2009: fig. 4). In the multi-module approach, using pL-systems and sets of productions, a structure is represented by a string of multi-modules, which consist of a collection of modules (B, shown with two modules per multi-module). As the multi-modules are the same across the whole string, the variables from within the individual modules are related to each other as before in the regular L-system, but this is only possible because the L+C language was extended to include separate lists of modules to consider/ignore for each L+C group. Thus, two (or more aspects) of the developing structure are represented independently by the approach, and, in addition, aspect weaving can occur through modules within a multi-module.

L-system-based modelling platform L-studio (Prusinkiewicz *et al.*, 2000*b*; Prusinkiewicz, 2004), where modules in the L-system represented various components of the shoot. In L+C, these modules are defined as

```
module Apex (int node, float vigour);
module AxBud (int node, int order);
module Internode (int node, float length,
  float radius);
module Leaf (int node, float area);
module Fruit (int node, float volume);
```

All these modules have in common a parameter for the node number along the shoot, but each module also has a parameter describing an attribute of a shoot organ. These attributes characterize the vigour of an apex, order of an axillary bud (1 = parent cane, 2 = axillary shoot), length and radius of an internode, area of a leaf, and volume of a fruit. The development of a kiwifruit shoot begins with an axillary bud specified in the axiom

```
axiom: AxBud(1,1)
```

with the node number set to 1, and the parent cane assumed to have order 1. Rewriting rules for the production of organs are specified in an L+C group, DEVELOP:

```
group DEVELOP:
Apex(node,vigour): {
  if (... produce new metamer)
   produce Internode(node,0,0)
      SB() Leaf(node,0) EB()
      SB() AxBud(node,2) EB()
      Apex(node+1,vigour);
  else if (... shoot tip aborts)
    produce Apex(node,0);
  else
    produce Apex(node,vigour);
}
AxBud(node,order): {
  if (order == 1) // if parent cane
    produce ... preformed metamers
      Apex(1, SetVigour (node));
  else if (order == 2) { // if axillary shoot
    if (... reproductive bud)
      produce Fruit(node,0);
    else if (... dormant bud)
      produce AxBud(node,order);
    else // an axillary shoot starts to grow
      produce Apex(1,SetVigour(node));
  }
}
Internode(node,length,radius): {
  ... update length and radius
  produce Internode(node,length,radius);
}
Leaf(node,area): {
  ... update area
  produce Leaf(node,area);
}
Fruit(node,volume): {
  ... update volume
  produce Fruit(volume);
}
```

According to the first rule, an apex may produce a new metamer, abort or remain dormant, depending on certain conditions (e.g. its age). A metamer consists of an internode, a leaf and an axillary bud. This bud will produce a number of preformed metamers ending with a terminal apex, with the vigour of the apex set by a user-defined function based on the axillary bud's node number. An axillary bud on an axillary shoot is reproductive and may produce a fruit if its node number is between 6 and 12; otherwise, it will remain dormant until apical dominance is lost, in which case it will start to grow. The growth of internodes, leaves and fruit could be defined empirically, but, instead, let us incorporate an aspect that models the growth of an organ based on carbon availability.

Carbon dynamics aspect

Prusinkiewicz *et al.* (2007*a*) presented an algorithm for simulating the acquisition, transport and partitioning of carbon within a plant based on an analogy between pressure-driven flow and current flow in an electric circuit, originally developed for a FSPM of a peach tree (Allen *et al.*, 2005). The L-system implementation of this carbon transport-resistance allocation model (C-TRAM) is compact and shows the benefit of L-system-based models: the automatic updating of the system of equations (representing flow of carbon in the plant) as the structure develops. The essence of C-TRAM is to compute the concentration and flow rates of all the sinks and sources within the plant based on the resistances between them. This allows the estimation of the carbon content of a single sink or source over time, given an initial value. For example, C-TRAM can be used to solve the following equation for a sink:

$$\frac{\mathrm{d}s}{\mathrm{d}t} = f(c, \ldots)G_{\max}(\ldots) \tag{1}$$

where ds/dt is the flow of carbon into the sink, c is the concentration outside the sink in the transport pathway and s is the carbon content of the sink. The function $f(c, \ldots)$ captures the sink's response to resource limitation, and $G_{\max}(\ldots)$ is the sink's maximum potential growth rate. These two functions are set by the user and were described for kiwifruit by Cieslak *et al.* (2011).

In the L+C implementation of C-TRAM, the concentrations and flow rates are computed in three phases, corresponding to three L+C groups. In the first phase, the system of equations representing the entire branching structure is reduced to a single equation that characterizes carbon concentration at its base. In the second phase, carbon concentrations in the remaining metamers are calculated in succession, given the values found for the neighbouring metamers. Finally, in the third phase, the distribution of concentrations is used to update the values of carbon flow. The first phase was described as *folding* and the second as *unfolding* by Prusinkiewicz *et al.* (2007*a*), because, metaphorically speaking, the network of sink/source elements is folded into one element, then unfolded back into the original network. A straightforward application of this computation scheme is only possible when $f(c, \ldots)$ is a linear function of c; otherwise, if $f(c, \ldots)$ is non-linear, another step must be implemented to solve the equations using linear approximations (the Newton–Raphson method). In that case, there are four phases, namely linearization, folding, unfolding and update flow, which are iterated, until the cumulative error in the carbon flow from all sources to all sinks falls below a user-defined threshold. See Prusinkiewicz *et al.* (2007*a*) for a detailed description of this process.

In the shoot model presented here, the definition of a module describing a sink or source is given by the following L+C code:

```
struct SinkSourceData {
  // carbon concentration outside
  // the sink/source
  float c;
  // flow of carbon into the sink or
  //out of the source
  float dsdt;
  // the carbon content of the sink/source
  float s;
  // resistance to flow past the sink/source
  float r;
//...other variables needed for computation
}
module S(SinkSourceData);
```

This definition differs from that given by Prusinkiewicz *et al.* (2007*a*), because the names of the members of the data structure are not given in terms of electric circuits (e.g. `float c;` for carbon concentration instead of `float v;` for voltage potential in the circuit), and the physical and physiological characteristics of a metamer are not included in its definition. Ultimately, this module will be used to represent a part of an organ's physiology, such as a growth sink or a maintenance respiration sink.

A skeleton of the L-system rules for folding and unfolding is given next to provide an idea of how C-TRAM works. The point is to show that these rules can be incorporated into the shoot model almost without changes from their original specification (Prusinkiewicz *et al.*, 2007*a*). After linearizing the equations (e.g. those of the form given in eqn 1), the algorithm scans the L-system string from right to left and applies the L-system rules to fold the branching structure. These rules are contained within the `FOLDING` group:

```
group FOLDING:
S(sd) >> SB() S(sdr2) EB() S(sdr1): {
  ...simplify equations describing a branching point with
     three elements, storing the result in sd, where sdr2 is
     a branching element enclosed by the SB (start
     branch) and EB (end branch) modules
produce S(sd);
}
S(sd) >> SB() S(sdr) EB(): {
  ...simplify equations describing a branching point with
     two elements, storing the result in sd, where sdr is a
     branching element
produce S(sd);
}
S(sd) >> S(sdr): {
  ...simplify equations describing two non-branching
     elements storing the result in sd
  produce S(sd);
}
S(sd): {
  ...start folding process (no need to simplify equation)
  produce S(sd);
}
```

The first two rules handle branching points with three or two elements, respectively. The third rule handles consecutive elements that do not include a branching point. These three rules simplify equations describing several sink/source elements using theorems for calculating equivalent circuits in a linear electric circuit (Prusinkiewicz *et al.*, 2007*a*). The last rule applies only to the most distal element of each axis and it is not necessary to simplify the equations.

After the folding step, C-TRAM calculates the carbon concentrations of each sink/source element by unfolding the branching structure from left to right. The corresponding rules form the UNFOLDING group:

```
group UNFOLDING:
S(sdl) << S(sd): {
  ...compute carbon concentration of current element,
     sd, given sink/source element to the left, sdl
  produce S(sd);
}
S(sd): {
  ...start unfolding process
  produce S(sd);
}
```

The first rule handles consecutive elements in the string, while the second rule is applied only to the very first element in the string. A branching rule is not necessary, as left context in an L-system is matched irrespective of the possible branching points in the structure. The carbon concentration of each sink/source element is computed given the carbon concentration from its left neighbour using theorems for calculating equivalent circuits in a linear electric circuit, similar to the folding process (Prusinkiewicz *et al.*, 2007*a*).

Finally, the carbon flow equations are updated given the newly computed carbon concentrations (Prusinkiewicz *et al.*, 2007*a*):

```
group UPDATEFLOW:
  S(sd):{
    ...update carbon flow for this sink/source element, sd
    ...calculate difference between previous and current
       flows
    ...accumulate this difference in a global variable
       error
    produce S(sd);
}
```

where the globally defined error variable is initialized to zero before the folding process starts, and represents the sum of the absolute values of differences between the previous and current flows from all elements. If the error is not sufficiently small, the carbon concentrations and flows of the sink/source elements are re-estimated in another iteration of the algorithm using the newly updated values, thus improving the estimate.

Now, the challenge is to integrate C-TRAM with the kiwifruit shoot model without making any significant changes to the algorithm. In particular, we do not want to change the SinkSourceData structure, so that the rules for folding and unfolding do not have to change.

Integration of architecture and carbon dynamics

The original L-system implementation of the carbon transport-resistance algorithm proposed by Allen *et al.* (2005) uses one module to represent a metamer. There are two shortcomings to this approach: (1) the data structure representing the metamer contains all possible members describing features of different organ types, even if the organ is not present, and (2) the representation of a metamer with more than one sink or source element is not possible, unless the data structure is changed and the rules for folding and unfolding are changed to account for this. This second drawback is especially difficult to address if any of the elements are represented by non-linear equations, as a suitable numerical method would be required to solve them. That is, it would require the application of the Newton–Raphson method on the elements within the metamer itself before application of the same method to all the metamers.

Here, we use the multi-module approach to create a kiwifruit shoot model with two aspects: one that handles the growth and development of organs, and another that handles the carbon allocation between sinks and sources. Specifically, the module Internode(), representing an internode, and the module S(), representing a sink/source, are combined to form a multi-module Internode() S(). One benefit is that any number of S() modules can be combined with an Internode() module to form a multi-module representing several aspects of the organ's physiology (e.g. physical properties and growth sink).

To integrate C-TRAM with the kiwifruit shoot model, pL-systems are used to combine S() modules with those already described in the paper (Internode(), Leaf(), etc.). In the axiom, the axillary bud on a parent cane is combined with a sink element:

```
axiom: AxBud(1,1) S(sdInit);
```

where sdInit is as an initial value for the SinkSourceData structure. Next, the S() modules must be integrated with the development of the shoot structure, because if an apex produces a new metamer, the appropriate sinks/sources must also be produced, and the growth of organs must be based on carbon availability. The L+C group, DEVELOP, changes to

```
group DEVELOP:
Apex(node,vigour) S(sd): {
  if (...produce new metamer) {
    ...initialize sink/source data
    ...for an internode, set sdI1, sdI2, and sdI3
    ...for a leaf, set sdL1, sdL2, and sdL3
    produce Internode(node,0,0)
      S(sdI1) S(sdI2) S(sdI3)
      SB() Leaf(node,0) S(sdL1) S(sdL2)
        S(sdL3) EB()
      S(sdInit)
      SB() AxBud(node,2) S(sdInit) EB()
      Apex(node+1,vigour) S(sd);
  }
  else
    produce Apex(node,vigour) S(sd);
}
AxBud(node,order) S(sd): {
  ...same rules as in previous specification of this rule,
      except for the addition of S() modules following Fruit()
        and Apex()
  produce AxBud(node,order) S(sd);
}
Internode(node,length,radius)
S(sd1) S(sd2) S(sd3): {
  ...update internode sink size, sd1, and resistance, sd1.r
  ...update internode maintenance, sd2
  ...update internode reserves, sd3
  ...update internode length and radius based on sink size,
     sd1
  produce Internode(node,length,radius)
    S(sd1) S(sd2) S(sd3);
}
Leaf(node,area) S(sd1) S(sd2) S(sd3): {
  ...update leaf sink size, sd1
  ...update leaf maintenance sink, sd2
  ...update carbon acquired through photosynthesis, sd3
  ...update leaf area based on sink size, sd1
  produce Leaf(node,area)
    S(sd1) S(sd2) S(sd3);
}
Fruit(node,volume) S(sd1) S(sd2): {
  ...update fruit sink size, sd1
  ...update fruit maintenance sink, sd2
  ...update fruit volume base on sink size, sd1
  produce Fruit(node,volume)
    S(sd1) S(sd2);
}
```

The major change here is that all the rules in this L+C group no longer use a strict predecessor, and the number of `S()` modules depends on the number of sinks/sources associated with an organ. To simplify the description, the apex and axillary bud do not accumulate carbon, so the only changes to the rules involving those organs are the addition of sink/source elements. For example, if the `Apex()` module produces a new metamer within a derivation step, the appropriate sink/source elements must also be produced. The internode is coupled with two sinks, one for growth and another for maintenance respiration, and one sink-source element for synthesis and hydrolysis of carbon reserves. The leaf is coupled with a sink for growth and another for maintenance respiration and a source of carbon from photosynthesis. Resistance to carbon flow between the various sinks and sources of a single metamer is set to zero, but resistance to flow throughout the whole shoot structure is set by the internodes. This is done by setting the `r` member of the `SinkSourceData` structure in the first sink of an internode, according to its length and radius. The last three rules are used to update the amount of carbon accumulated in the sinks and the amount remaining in the sources. This is done by applying the forward Euler method to numerically integrate the carbon flow into a sink or out of a source. For a sink/source element with the associated data structure `sd`, the integration is written as

```
sd.s=sd.s+sd.dsdt * DT;
```

where `DT` is the time step size. Its value is set by the user as a compromise between the accuracy and speed of computation. Finally, to handle multiple branching points for the folding/unfolding steps of C-TRAM, an inert sink is placed in between branches, e.g. `SB()...EB() S(sdInit) SB()...EB()`, as this is the simplest way to handle the branching point.

The next step in integrating C-TRAM is to include the folding and unfolding phases of the algorithm. By specifying the L+C command

```
consider: S;
```

at the beginning of the `FOLDING` and `UNFOLDING` groups, only `S()` modules will be matched in context-sensitive rules. In that way these two groups, which have already been described, do not change at all, as L+C will ignore all modules except `S()` in rule matching. More importantly, solving the non-linear equations for carbon flow requires only a single application of the Newton–Raphson method for all sinks/sources in the model.

The final step of integration is to linearize the equations representing flow of carbon into a sink or out of a source. This is done in a new group, `LINEARISE`, which updates the equations for carbon flow depending on sink/source type. It is necessary to have a separate linearize phase, so that information exchange can occur between sinks/sources at the local level within one plant component (e.g. a maintenance sink can query a primary growth sink for its size). Prusinkiewicz *et al.* (2007*a*) give the technique for linearizing these equations.

To complete the integration of C-TRAM into the kiwifruit shoot model, the `UPDATEFLOW` group can be added without any changes. Figure 2 shows the flow of computational phases for the whole model presented in this section. There are five phases implemented in L+C groups: the `DEVELOP` group handles production of new metamers and growth of organs based on carbon availability, the `LINEARISE` group begins the application of the Newton–Raphson method by linearizing the carbon flow equations for sinks and sources, and the `FOLD`, `UNFOLD` and `UPDATEFLOW` groups solve the

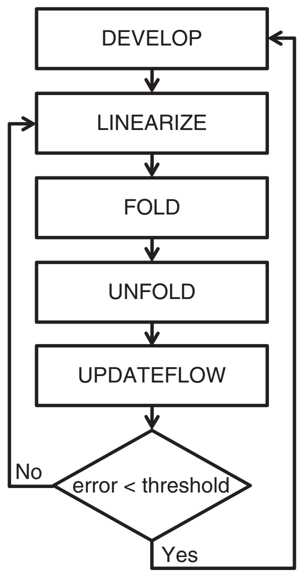

FIG. 2. The flow of computational phases for the kiwifruit shoot model, combining development of the shoot with C-TRAM. Each phase is implemented as an L+C group, with one phase applied during an L-system derivation step. Rule matching in each phase is done by scanning the string from left to right, except in the folding phase, where scanning is from right to left.

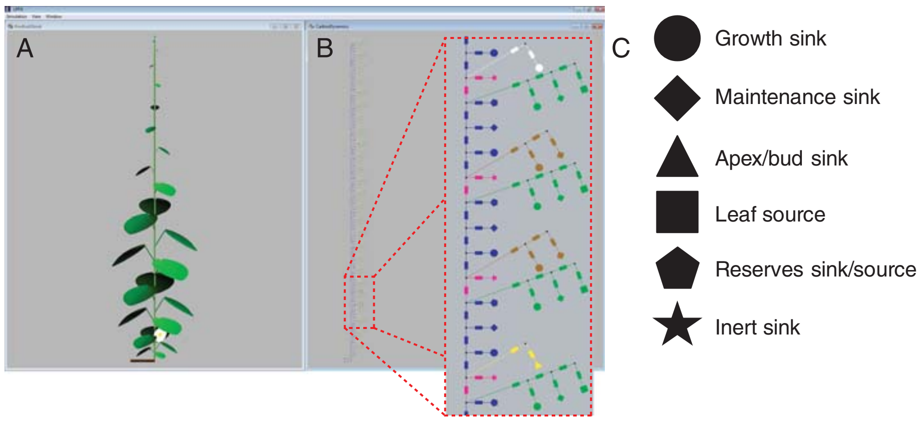

FIG. 3. Visualization of the kiwifruit shoot model with two aspects: (A) architectural and (B) carbon dynamics. A close-up of the sinks/sources of the first six metamers is shown in the inset. Elements belonging to an internode are coloured blue, to a leaf are green, to a fruit are brown, to a flower are white, to an apex or axillary bud are yellow, and to an inert element are magenta. The different types of sinks/sources are represented by the symbols shown in (C).

system of equations representing the carbon concentration and flow throughout the structure. After application of the UPDATEFLOW group, the accumulated error of the linearization is checked, and if it is within a specified threshold, the development of the shoot continues; otherwise, the Newton–Raphson method is applied again.

Figure 3 shows a visualization of the model with two aspects. The L-system rules in the DEVELOP group apply to the modules defined for the growth and development aspect (visualized in Fig. 3A) and the rules for C-TRAM apply to the modules defined for the carbon allocation aspect (visualized in Fig. 3B). Now, let us investigate the capabilities of this approach even further by extending the shoot model with another aspect.

Adding a signalling mechanism

To show how the kiwifruit shoot model can be extended with a new aspect, let us consider the process of apical dominance in the shoot, specifically the inhibition of lateral bud activation by the shoot apex, and the outgrowth of axillary buds upon removal of the apex (Cline, 1991). Different hypotheses have been proposed to account for the effect of apical dominance on the control of branching [addressed by Dun *et al.* (2009) from a modelling prospective]. One of these hypotheses, the auxin transport hypothesis, proposes that active basipetal transport of the plant hormone auxin down the main stem controls outgrowth of buds through competition between apices for auxin transport in the main stem (Bennett *et al.*, 2006).

Based on this hypothesis, Prusinkiewicz *et al.* (2009) developed a computational model to explain apical dominance using an auxin transport switch. In their model bud activation is regulated by auxin flux; if the auxin efflux from a lateral bud surpasses a fixed threshold, the bud becomes active. This relies on the positive feedback between auxin flux and PIN protein polarization of active auxin transport, and assumes that the interplay between them can be integrated from the cellular level to the metamer level (Sachs, 1991). Prusinkiewicz *et al.* (2009) presented auxin flux from metamer i to metamer j as the following equation:

$$\Phi_{i\to j} = Ta_i[PIN_{i\to j}] - Ta_j[PIN_{j\to i}] + D(a_i - a_j) \qquad (2)$$

where a_i is auxin concentration, $PIN_{i\to j}$ is the concentration of PIN proteins actively transporting auxin from metamer i to j, T is a polar transport coefficient and D is a diffusion coefficient. The change in $PIN_{i\to j}$ concentration depends on the flux $\Phi_{i\to j}$, where auxin flux drives PIN allocation to the face of metamer i neighbouring metamer j. It was given as

$$\frac{\mathrm{d}[PIN_{i\to j}]}{\mathrm{d}t} = \begin{cases} \rho_{i\to j}\dfrac{\Phi^n_{i\to j}}{K^n + \Phi^n_{i\to j}} + \rho_0 - \mu[PIN_{i\to j}] & \Phi_{i\to j} \geq 0 \\ \rho_0 - \mu[PIN_{i\to j}] & \Phi_{i\to j} < 0 \end{cases} \qquad (3)$$

where $\rho_{i\to j}$ is the maximum PIN allocation rate dependent on auxin flux, ρ_0 is the base PIN allocation rate independent of auxin flux, μ is PIN turnover rate, and K and n are coefficients in the Hill function used to capture PIN allocation (Prusinkiewicz *et al.*, 2009). The change in auxin concentration within metamer i is given by

$$\frac{\mathrm{d}a_i}{\mathrm{d}t} = \sum_j \Phi_{j\to i} + \sigma(H - a_i) - va_i \qquad (4)$$

where σ is the auxin production rate, H is the target auxin concentration and v is the auxin turnover rate. This equation is slightly different from the one presented by Prusinkiewicz *et al.* (2009), because the metamer volume and area of the face adjoining two metamers are not explicitly included in the equation. In this version, both of these values are just assumed to be equal to one. Lastly, the model assumes auxin production occurs only within terminal and lateral apices (otherwise, $\sigma = 0$).

As this apical dominance model will be integrated into the kiwifruit shoot model, it is easier to model the flow of auxin from apices through internodes instead of through metamers, as metamers are not explicitly included in the kiwifruit shoot model. First, two modules that handle auxin transport are defined as follows:

```
struct AuxinData {
  float a; // auxin concentration
};
struct WallData {
  // auxin flux out of an apex/internode
  float flux;
  // PIN concentration on wall i to j
  float PINi;
  // PIN concentration on wall j to i
  float PINj;
};
module A(AuxinData ad);
module W(WallData wd);
```

where `ad` is a variable of the `AuxinData` structure and `wd` is a variable of the `WallData` structure. Both of these modules are associated with the apex and internode components, and operate as a pair. The `W` modules, which separate two `A` modules, are used to calculate auxin flux due to diffusion and basipetal polar transport down a shoot, as defined in the following L+C group:

```
group AUXINFLUXES:
consider: A;
A(adL)<W(wd)>A(adR): {
    ... compute diffusion and polar transport through this wall
    ... update auxin flux and PIN concentration
    produce W(wd);
}
```

The flux computed by the `W` modules is then used to update the auxin concentration in the `A` modules as follows:

```
group AUXINTRANSPORT:
consider: W;
W(wdL)<A(ad)>SB() EB()
  SB() W(wdLAT) EB() W(wdR): {
    ... update auxin concentration according to flux
    produce A(ad);
}
W(wdL)<A(ad)>SB() EB()
  SB() W(wdLAT) EB(): {
    ... same as above but without the right neighbour
    produce A(ad);
}
W(wdL)<A(ad)>W(wdR): {
    ... same as above but without the lateral neighbour
    produce A(ad);
}
W(wdL)<A(ad): {
    ... same as above but with only the left neighbour
    produce A(ad);
}
A(ad)>W(wdR): {
    ... same as above but with only the right neighbour
    produce A(ad);
}
```

In all of these rules, a module's left neighbour corresponds to the first adjoining component (internode, apex or axillary bud) in the basipetal direction and its right neighbour corresponds to the first adjoining component in the acropetal direction along the shoot. There are five possibilities that must be handled by the rules in this group: the first rule applies to the modules that have left, right and lateral neighbours; the second rule applies to modules with left and lateral neighbours (e.g. when the shoot is pruned, this matches the first internode immediately below the cut); the third rule applies to modules with no lateral neighbours; the fourth rule applies to terminal or lateral modules (where the shoot apices are located); and the fifth rule only applies to the first module in the string. Within each rule, the auxin concentration of each component is updated by numerically solving eqn (4) using the forward Euler method with a time step of 0·1. Note, in the first two rules, an extra set of branch modules (`SB() EB()`) enclosing a leaf is included to ensure a match at the branching point.

This model of auxin flow from apices through internodes, implemented in the L+C group `AUXINFLOW`, can be integrated into the kiwifruit shoot model by placing `A()` and `W()` modules after each `Apex()`, `AxBud()` and `Internode()` module. First, the axiom is updated to

```
axiom: AxBud(1,1) S(sdInit)
    A(adInit) W(wdInit);
```

where the variables `adInit` and `wdInit` initialize the auxin data. Then, the control flow of computational phases (shown in Fig. 2) is changed to include the new `AUXINFLOW` and `AUXINTRANSPORT` groups (see Fig. 4). This implementation assumes that the time step for the carbon dynamics and auxin flow aspects are the same (i.e. $\mathrm{DT} = 0{\cdot}1$), but this is only for convenience. In order to have different time steps, an additional L+C group could be implemented that performs the forward Euler step for each aspect. Using a common time step, however, only requires updating the `DEVELOP` group by placing `A()` and `W()` modules within the predecessors of the rules involving the three related modules, as follows:

```
group DEVELOP:
Apex(node,vigour) S(sd) A(ad) W(wd): {
  ... same calculations as in previous declaration
    but, in addition, update production of auxin, ad.a
  if (... produce new metamer) {
    ... initialize auxin data for lateral, adInit and wdInit
    produce Internode(node,0,0)
      S(sdI1) S(sdI2) S(sdI3)
      A(ad) W(wd)
      SB() Leaf(node,0) S(sdL1) S(sdL2)
        S(sdL3) EB()
      S(sdInit)
      SB() AxBud(node,2) S(sdInit)
        A(adInit) W(wdInit) EB()
      Apex(node+1,vigour) S(sd)
        A(ad) W(wd);
  }
  else
    produce Apex(node,vigour) S(sd)
      A(ad) W(wd);
```

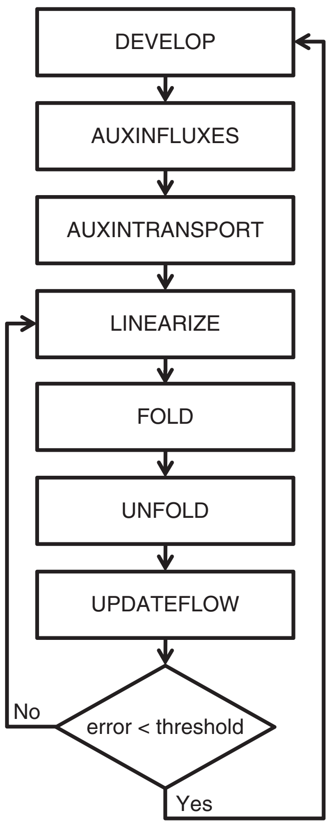

FIG. 4. The flow of computational phases for the kiwifruit shoot model, updated to include control of branching by apical dominance. In addition to updating the amount of accumulated carbon per sink, the DEVELOP phase is used to compute the production of auxin in apices and to check for lateral bud activation. The AUXINFLUXES phase computes auxin diffusion and active basipetal transport through the shoot, and PIN concentration. The AUXINTRANSPORT phase computes auxin concentration based on flux.

```
}
AxBud(node,order) S(sd) A(ad) W(wd): {
  ...same as before, except for the following change:
  if (wd.flux>EFFLUX_THRESHOLD)
    produce Apex(node,1) S(sd) A(ad) W(wd);
  else
    produce AxBud(node,order)
      S(sd) A(ad) W(wd);
}
Internode(node,length,radius)
S(sd1) S(sd2) S(sd3) A(ad) W(wd): {
  ...update carbon dynamics as before
  produce Internode(node,length,radius)
        S(sd1) S(sd2) S(sd3) A(ad) W(wd);
}
...the remaining rules do not change
```

When an apex produces a new metamer, an `A()` and `W()` module is placed following the `Internode()` module and the variable containing the auxin data is copied from the apex (so the auxin and PIN concentrations remain the same). A new `A()` and `W()` module is placed following the axillary bud with the auxin data initialized using the variable `adInit` and `wdInit`, respectively. The most substantial change is within the rule for axillary bud development, where if the auxin efflux is greater than some predefined threshold, the bud is activated and will start to produce new metamers.

Figure 5 shows a visualization of the kiwifruit shoot model and the apical dominance model. To show how decapitation of the shoot activates a lateral, several steps of the simulation are shown over time. The model parameters for auxin flow were set exactly as in the simulation of decapitation experiments by Prusinkiewicz *et al.* (2009: fig. 2G, H), but the threshold for bud activation was set to 6·7 instead of 2. The reason for the difference in values was to ensure that only one lateral grew after pruning the shoot, as has been observed for kiwifruit shoots (Minchin *et al.*, 2010).

Adding biomechanics for shoot bending

Considering a model of shoot mechanics proposed by Fournier *et al.* (1994), who treated woody stems as elastic rods subject to primary and secondary growth, Jirasek *et al.* (2000) developed an L-system implementation of a biomechanical plant model that simulates bending and twisting of branches. A more efficient L+C implementation using fast information transfer was subsequently devised by Taylor-Hell (2005). Prusinkiewicz *et al.* (2007*b*) distilled the essence of this model in an L+C example operating in two dimensions. We use this implementation to show how easily a previously devised biomechanics aspect can be incorporated into a comprehensive shoot model using the proposed multi-module approach. We also show how information is exchanged between aspects: in particular, how the length and mass of the shoot (determined by carbon dynamics) are passed to the biomechanics aspect, affecting the shape of the shoot. The model can be extended to three dimensions (Costes *et al.*, 2008), by replacing mathematical details of the biomechanical aspect while preserving the structure of the integrated model.

Prusinkiewicz *et al.* (2007*b*) modelled a shoot (branch axis) as a sequence of rigid internodes connected at elastic nodes. Each internode is represented by a vector $\vec{r}_i = s_i\vec{H}_i$, where i is the internode number along the shoot, s_i is the internode length and $\vec{H}_i$ is a unit vector giving the internode's orientation. The mass of each internode is concentrated at its distal node. Each node is subject to a torque caused by gravity acting on the shoot. The shape of the shoot in static equilibrium is calculated using a relaxation method (Press *et al.*, 1992) that consists of iterative application of two steps: (1) given $\vec{r}_i$ calculate the torque, $\vec{\tau}_i$, at node i due to gravity acting on the nodes distal to it, $j > i$, and (2) given $\vec{\tau}_i$ calculate the new orientation $\vec{H}_i$ and update $\vec{r}_i$. These two steps are repeated until the accumulated changes in the orientations of all internodes are small, at which point it is assumed that an equilibrium state has been found.

The torque, $\vec{\tau}_i$, is calculated using the following equation (Prusinkiewicz *et al.*, 2007b):

$$\vec{\tau}_i = \vec{\tau}_{i+1} + \vec{r}_{i+1} \times \sum_{j=i+1}^{N} \vec{F}_j \tag{5}$$

where N is the total number of nodes in the shoot, and $\vec{F}_j = m_j\vec{g}$ is the force of gravity acting on node j with mass

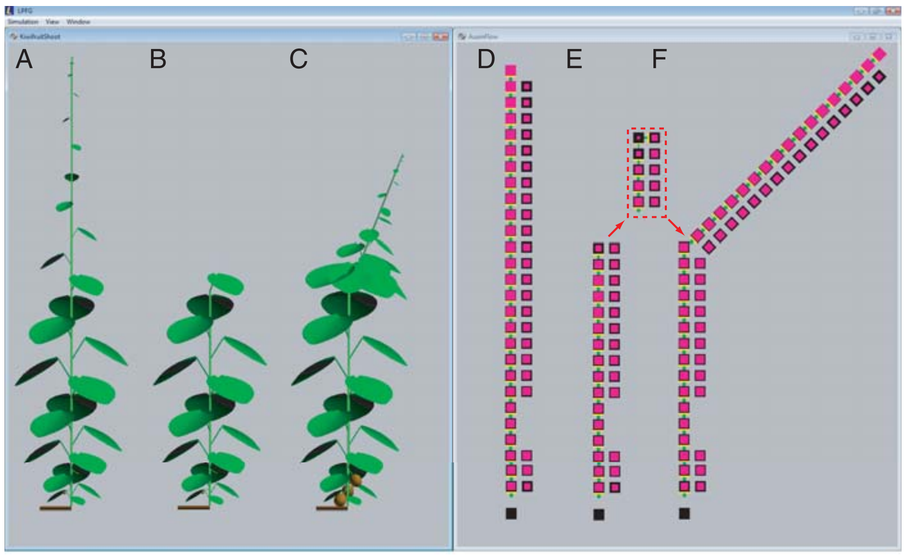

FIG. 5. Visualization of the kiwifruit shoot model integrated with an apical dominance aspect. The image sequence on the left shows the development of the shoot, and the one on the right shows auxin flow from apices through internodes (both components are visualized as squares). The schematic representation of auxin flow is based on the one used by Prusinkiewicz *et al.* (2009), where the edge length of the magenta square gives auxin concentration in the component, the width of the yellow rectangle gives PIN concentration in the adjoining component face, and the width of the green rectangle gives the auxin flux out of a component. There are three time points shown: (A, D) the shoot before decapitation, (B, E) the shoot immediately after decapitation, and (C, F) the development and growth of a lateral. The inset (dashed-red rectangle) shows the state of the lateral bud just before activation, when its auxin efflux starts increasing. Decapitation of the shoot was performed by interactively selecting (on the screen) the location of the cut. There are three lateral buds missing at nodes 4, 5 and 6 because fruit has already appeared at those positions. Finally, the black square at the bottom of the auxin flow visualization (D–F) represents the major auxin sink, which has a high auxin turnover rate.

m_j, given gravitational acceleration $\vec{g}$. This equation can be extended to calculate torques in the case of a branching shoot by adding torques from the lateral branches. For example, in the case of one branch at node i, the combined torque $\vec{\tau}_i^c$ is equal to

$$\vec{\tau}_i^c = \vec{\tau}_i + \vec{\tau}_i^b \tag{6}$$

where $\vec{\tau}_i$ and $\vec{\tau}_i^b$ represent the accumulated torques exerted on node i by the main shoot and the lateral branch, respectively. Each of these component torques is calculated according to eqn (5).

The orientation $\vec{H}_{i+1}$ of internode $\vec{r}_{i+1}$ is found by calculating its rotation with respect to internode $\vec{r}_i$, caused by the torque acting on node $i+1$. In principle, the angle of rotation caused by torque $\vec{\tau}_{i+1}^c$ is equal to $\alpha_{i+1} = \tau_{i+1}^c/\kappa_{i+1}$, where τ_{i+1}^c is the magnitude of torque $\vec{\tau}_{i+1}^c$, and κ_{i+1} is a rotational spring constant associated with node $i+1$. However, as changes in the orientations of internodes also change the torques, the equilibrium configuration of the branch is found iteratively, using a relaxation method. To this end, in each iteration step the orientation $\vec{H}_{i+1}$ is adjusted using the equations:

$$\begin{aligned}\vec{R} &= \vec{H}_{i+1} + k(\vec{H}_{i+1}^{\text{previous}} - \vec{H}_{i+1})\\ \vec{H}_{i+1}^{\text{adjusted}} &= \vec{R}/\|\vec{R}\|\end{aligned} \tag{7}$$

where $\vec{H}_{i+1}^{\text{previous}}$ is the orientation of internode $\vec{r}_{i+1}$ found in the previous iteration step, the temporary variable $\vec{R}$ represents the adjusted orientation before normalization, $\vec{H}_{i+1}^{\text{adjusted}}$ is the normalized adjusted orientation resulting from the current relaxation step, and parameter k controls the speed of convergence to the solution. The relaxation proceeds until the differences between previous and calculated orientations, $\left\|\vec{H}_{i+1}^{\text{previous}} - \vec{H}_{i+1}\right\|$, summed over all nodes i, becomes sufficiently small, at which point an equilibrium is assumed to have been found. Finally, positions $\vec{P}_i$ of all nodes are found using the recursive formula $\vec{P}_{i+1} = \vec{P}_i + s_i\vec{H}_i$.

This biomechanical model can be integrated into the kiwifruit shoot model in the same way as the carbon dynamics model was integrated. First, a data structure and module are defined that represent an internode from a biomechanical perspective:

```
struct BiomechanicsData {
  float s; // length of internode
  float mass; // mass of the node
  float ke; // elasticity at a node
  float torque; // combined torque at the node
  V3f P; // position at the node
  V3f H; // orientation of the internode
  // ... other variables needed for computation
  };
module B(BiomechanicsData);
```

where `V3f` is a user-defined data type representing a three-dimensional vector. We assume the most basal node is at point (0,0,0) and all the other nodes are relative to their supporting node. Now, `B()` modules are placed following the `Internode()` modules, and the members defined in the `BiomechanicsData` structure can be updated by a pL-system rule, as was done for the sinks/sources in the carbon dynamics aspect. The orientations of the internodes are computed by calculating torques in one L+C group and rotations in another, using eqns (5)–(7).

Jirasek *et al.* (2000) noticed that the forces and torques involved in shoot bending can be considered as signals that propagate basipetally down the shoot. Using this idea, Taylor-Hell (2005) and Prusinkiewicz *et al.* (2007*b*) implemented an L+C group that accumulates the masses and torques along a shoot using fast information transfer. The L+C group may be implemented in the following way, where the L-system string is derived backwards from right to left:

```
group PROPAGATELEFT:
consider: B;
B(bm) >> SB() EB()
  SB() B(bmLAT) EB() B(bmR): {
  ...sum mass from lateral and right neighbours
  ...accumulate torque due to gravity from lateral and
     right neighbours
  produce B(bm);
}
B(bm) >> SB() EB() SB() B(bmLAT) EB(): {
  ...sum mass from lateral neighbour
  ...accumulate torque due to gravity from lateral
     neighbour
  produce B(bm);
}
B(bm) >> B(bmR): {
  ...sum mass from right neighbour
  ...accumulate torque due to gravity from right neighbour
  produce B(bm);
}
B(bm): {
  ...set torque to zero
  produce B(bm);
}
```

The first rule applies to an internode with another one above it and a lateral branch at its node, i.e. a `B()` module with a right and lateral neighbour. Note, the extra pair of `SB() EB()` modules are necessary for the rule to be matched, as these enclose a `Leaf()` module and cannot be ignored in *lpfg*. The second rule applies to an internode with a lateral branch at its node but no internode above it, and the third rule is the opposite, with an internode above but no lateral branch. The last rule applies to the apical internode on a shoot; it initializes the process of torque accumulation along the shoot.

Based on the torques computed in the `PROPAGATELEFT` group, the orientation of the nodes can be updated using fast information transfer as follows:

```
group PROPAGATERIGHT:
consider: B;
  B(bmL) << B(bm): {
    ...calculate rotation required for equilibrium corre-
       sponding to the accumulated torque, bm.torque
    ...calculate the difference between the current
       rotation and the newly calculated one
    ...apply relaxation method: adjust current rotation,
       bm.H, by a fraction of this difference
      (step towards equilibrium)
    ...update position, bm.P, given the left neighbour's
       position, bmL.P
    produce B(bm);
}
B(bm): {
    ...set the position as (0,0,0)
    produce B(bm);
}
```

The first rule updates the orientation of an internode based on the combined torque acting on its node, and the position of the node. The second rule sets the position of the most basal node, without a left neighbour, using a base position specified at the start of the simulation. The magnitudes of the difference between an internode's orientation and the orientation required to achieve equilibrium is accumulated into a global error variable, which is used to measure the adequacy of the solution.

The `PROPAGATELEFT` and `PROPAGATERIGHT` groups can be placed directly into the kiwifruit shoot model implementation, but the `DEVELOP` group must be updated with the addition of `B()` modules after the `Internode()` modules as follows:

```
group DEVELOP:
  Apex(node,vigour) S(sd) A(ad) W(wd): {
    ...same calculations as in previous declaration
    if (...produce new metamer) {
      ...initialize biomechanics data for internode, bmInit
      produce Internode(node,0,0)
          S(sdI1) S(sdI2) S(sdI3)
          A(ad) W(wd) B(bmInit)
      ...the rest is unchanged
}
Internode(node,length,radius)
S(sd1) S(sd2) S(sd3) A(ad) W(wd) B(bm): {
    ...update carbon dynamics, sd1, sd2, and sd3
    ...update internode length and radius based on sink
       size, sd1
    ...copy length to bm.s and set mass in bm.mass
    ...set elasticity in bm.ke
    produce Internode(node,length,radius)
        S(sd1) S(sd2) S(sd3)
        A(ad) W(wd) B(bm);

}
  ...the remaining rules do not change
```

The only change for the rule involving the `Apex()` module is the placement of a `B()` module following an `Internode()` module upon production of a new metamer. The rule for the `Internode()` module must now include the `B()` module in the predecessor, allowing the length and mass of the internode to be passed from the carbon dynamics aspect to the biomechanical aspect. This is an example of how the multi-module approach provides a means of information exchange between different aspects.

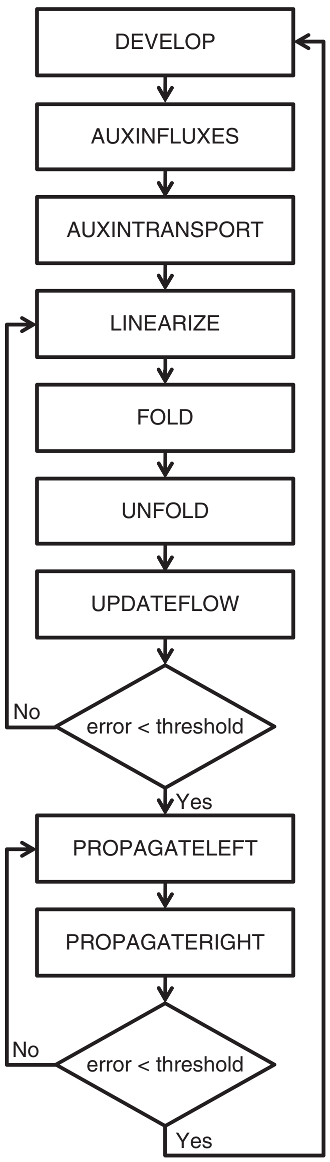

FIG. 6. The flow of computational phases for the kiwifruit shoot model, updated to include biomechanics for shoot bending. In addition to its other tasks, the DEVELOP group is now used to update the length and mass of each internode for the biomechanical aspect. The PROPAGATELEFT group accumulates (basipetally down the shoot) torques due to gravity acting on the nodes attached to the internodes. The PROPAGATERIGHT group adjusts the orientation of each internode towards an equilibrium state, which is calculated according to the torques and physical properties of the shoot. To this end, these two phases are repeated until an acceptable solution is found, i.e. until the accumulated change in the orientation of the internodes, within one time step, is small.

With the updated DEVELOP group, the integration of a biomechanical aspect into the kiwifruit shoot model is now complete. Figure 6 shows the updated flow of computational phases in the kiwifruit shoot model. The L+C groups involved in computation of the shoot biomechanics are run immediately after the carbon dynamics, but are still performed within the same number of time steps, as the length and mass of each internode are set in the DEVELOP group.

Figure 7 shows a visualization of the kiwifruit shoot and of its biomechanical representation. The initial orientation of the parent branch section that the shoot is growing from is set slightly downwards, so that the shoot will bend under the force of gravity. The acceleration due to gravity was $9{\cdot}81$ m s^{-2} and the parameter controlling the speed of convergence to the solution was $k = 0{\cdot}5$ (set as a compromise between the speed and accuracy of the solution). The elasticity of an internode was dependent on its age.

CONCLUSIONS

We have introduced an L-system-based modelling technique for integrating several aspects of plant function into well-structured, comprehensive FSPMs. The proposed technique is founded on the notion of multi-modules, which represent plant components using sequences of L-system symbols rather than single symbols. Each symbol within a multi-module corresponds to an aspect of the component's operation. L-system programming with multi-modules is supported by the following constructs:

(1) Pseudo-L-systems (Prusinkiewicz, 1986): the extension of L-systems that allows for the rewriting of multiple symbols by a single production.
(2) Groups, or the division of the production set into subsets, with a control mechanism deciding which subset is applicable in each simulation step (Dassow, 1986; Yokomori, 1986; Prusinkiewicz *et al.*, 2007*b*).
(3) Local consider/ignore statements, which make it possible to specify modules recognized for the purpose of context searching individually for each group.

Using these constructs, the user can specify which symbols will be affected in each simulation step, and thus select which functional aspects and/or architectural components will be of concern in this step. By dynamically changing the set of affected symbols from one step to another, criss-crossed concerns can be addressed in a modular manner.

The usefulness of the proposed technique was demonstrated by integrating previously modelled aspects of carbon dynamics (Allen *et al.*, 2005; Prusinkiewicz *et al.*, 2007*a*), apical dominance (Prusinkiewicz *et al.*, 2009) and biomechanics (Taylor-Hell, 2005; Prusinkiewicz *et al.*, 2007*b*) into a model of a developing kiwifruit shoot (see Fig. 8). These aspects were specified independently, and their implementation was based on the source code provided by the original authors. In the case of carbon dynamics, multi-modules allowed for the extension of components including a single sink or source to components with several sinks and sources without changing the underlying carbon allocation code. Our example thus shows that multi-modules provide a simple, practical technique for encapsulating aspects of plant function, composing them effectively into a comprehensive model, and, possibly, reusing them in other models.

Although, for the example presented here, integration of the aspects was performed manually, the multi-module approach could lead to an aspect-oriented programming language for building FSPMs, but would require the support of appropriate constructs. In particular, for the integration of

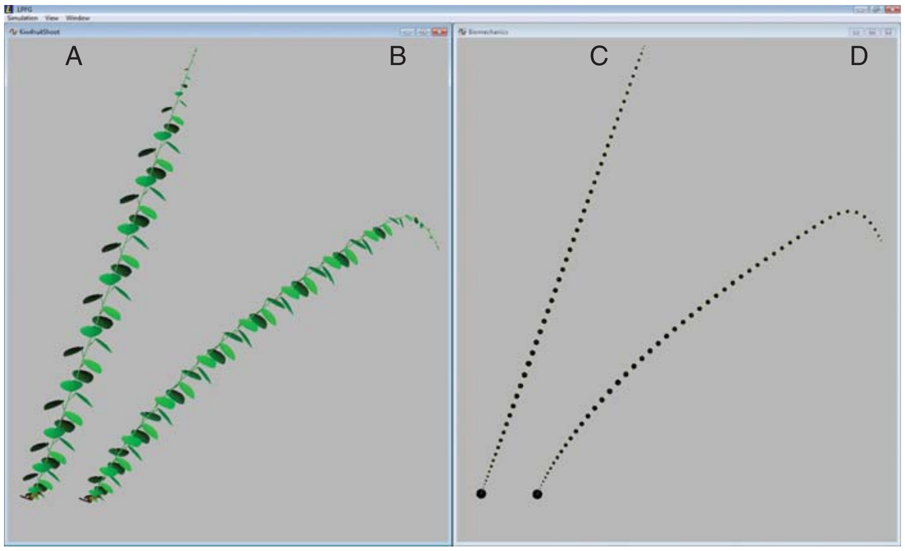

FIG. 7. Visualization of the kiwifruit shoot model integrated with a biomechanical aspect. It shows the kiwifruit shoot without (A, C) and with (B, D) bending due to gravity. On the right is the visualization of the biomechanical aspect, with an internode visualized as a yellow line attached to a node (black circle) at its distal end. The length of the line corresponds to the length of the internode and the size of the circle is proportional to the internode's mass (concentrated within the node). The orientation of the leaves due to tropism is simulated using *lpfg*'s built-in functionality (Karwowski and Lane, 2007).

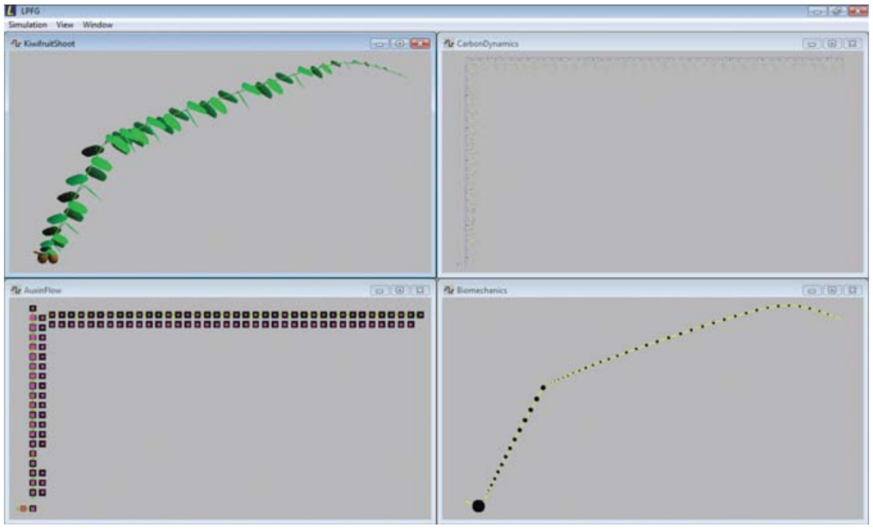

FIG. 8. Visualization of the kiwifruit shoot model integrated with four aspects of plant structure and function. From the top left, the first image shows a visualization of the kiwifruit shoot, the second image shows the sinks/sources, the third shows auxin flow from apices through the internodes, and the last one shows the biomechanics.

each aspect, it would require the automatic addition of the aspect's modules to the multi-modules and update of the flow of computational phases to include the groups implementing the aspect's production rules. This task may seem difficult in the case of aspects with a common time step, but it has already been addressed in other aspect-oriented languages, such as AspectC++ (Spinczyk *et al.*, 2005). Furthermore, even though the examples presented in this paper handled multiple branching points in a simple way, explicitly checking for them in the production rules, more sophisticated constructs from L+C could be used to handle the general case.

Another potential advantage of the multi-module approach is the specification of numerical methods using special computational modules, where one of the modules in a multi-module could be used for computation. This would allow for more efficient solving of systems of equations specified for developing structures, as there is no need to reformulate the equations and resubmit them to a standard solver each time they change, i.e. the solver would work on the topological space defined by the L-system (Prusinkiewicz, 2009). In addition, the implementation of the numerical method itself would be independent of the process, described by the system of equations, it was solving. For example, a numerical method for solving systems of non-linear equations in growing structures for two different processes, such as carbon dynamics and biomechanics, could be implemented using the same set of L-system productions. This has an advantage over the current approach in that the numerical method used to solve the non-linear equations for carbon dynamics and biomechanics would be formulated only once, instead of once for each aspect. This idea could be further applied to modelling signal propagation in plants, where a single production set could specify the flow of diverse signals that have a similar transport mechanism. To conclude, these types of applications of the multi-module approach illustrate its ability to handle the challenging problem of properly integrating several aspects of plant function into one model, and, importantly, the capability of the L-system formalism to do this.

ACKNOWLEDGMENTS

We thank Brendan Lane for discussions on the biomechanics aspect of the model, and Helge Dzierzon and the anonymous referees for useful appraisal of the manuscript. We gratefully acknowledge the support of this research provided by the Kiwifruit Royalty Investment Project, New Zealand Institute for Plant & Food Research Limited (M.C. and A.N.S.) and the Natural Sciences and Engineering Research Council of Canada (P.P.).

LITERATURE CITED

Allen MT, Prusinkiewicz P, DeJong TM. 2005. Using L-systems for modeling source-sink interactions, architecture and physiology of growing trees: the L-PEACH model. *New Phytologist* **166**: 869–880.

Bennett B, Sieberer T, Willett B, Booker J, Luschnig C, Leyser O. 2006. The Arabidopsis MAX pathway controls shoot branching by regulating auxin transport. *Current Biology* **16**: 553–563.

Bertheloot J, Andrieu B, Fournier C, Martre P. 2008. A process-based model to simulate nitrogen distribution in wheat (*Triticum aestivum*) during grain-filling. *Functional Plant Biology* **35**: 781–796.

Buck-Sorlin GH, Kniemeyer O, Kurth W. 2005. Barley morphology, genetics and hormonal regulation of internode elongation modelled by a relational growth grammar. *New Phytologist* **166**: 859–867.

Cieslak M, Seleznyova AN, Hanan J. 2011. A functional–structural kiwifruit vine model integrating architecture, carbon dynamics, and effects of the environment. *Annals of Botany* **107**: 747–764.

Cline MG. 1991. Apical dominance. *Botanical Review* **57**: 318–358.

Costes E, Smith C, Renton M, Guédon Y, Prusinkiewicz P, Godin C. 2008. MAppleT: simulation of apple tree development using mixed stochastic and biomechanical models. *Functional Plant Biology* **35**: 936–950.

Dassow J. 1986. On compound Lindenmayer systems. In: Rozenberg G, Salomaa A. eds. *The book of L.* Berlin: Springer-Verlag, 77–85.

Dun EA, Hanan J, Beveridge CA. 2009. Computational modeling and molecular physiology experiments reveal new insights into shoot branching in pea. *Plant Cell* **21**: 3459–3472.

Foster TM, Seleznyova AN, Barnett AM. 2007. Independent control of organogenesis and shoot tip abortion are key factors to developmental plasticity in kiwifruit (*Actinidia*). *Annals of Botany* **100**: 471–481.

Fourcaud T, Zhang X, Stokes A, Lambers H, Körner C. 2008. Plant growth modelling and applications: the increasing importance of plant architecture in growth models. *Annals of Botany* **101**: 1053–1063.

Fournier C, Andrieu B. 1999. ADEL-maize: an L-system based model for the integration of growth processes from the organ to the canopy. Application to regulation of morphogenesis by light availability. *Agronomie* **19**: 313–327.

Fournier M, Bailleres H, Chanson B. 1994. Tree biomechanics: growth, cumulative prestresses, and reorientations. *Biomimetics* **2**: 229–251.

Han L, Hanan J, Gresshoff PM. 2010. Computational complementation: a modelling approach to study signalling mechanisms during legume autoregulation of nodulation. *PLoS Computational Biology* **6**: e1000685. doi:10.1371/journal.pcbi.1000685.

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

Hanan J, Prusinkiewicz P, Zalucki M, Skirvin D. 2002. Simulation of insect movement with respect to plant architecture and morphogenesis. *Computers and Electronics in Agriculture* **35**: 255–269.

Hemmerling R, Kniemeyer O, Lanwert D, Kurth W, Buck-Sorlin GH. 2008. The rule-based language XL and the modelling environment GroIMP illustrated with simulated tree competition. *Functional Plant Biology* **35**: 739–750.

Jirasek C, Prusinkiewicz P, Moulia B. 2000. Integrating biomechanics into developmental plant models expressed using L-systems. In: Spatz HC, Speck T. eds. *Plant biomechanics 2000.* Stuttgart: Georg Thieme Verlag, 615–624.

Karwowski R, Lane B. 2007. *LPFG user's manual.* Calgary: University of Calgary,

Karwowski R, Prusinkiewicz P. 2003. Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science* **86**.

Kiczales G, Lamping J, Mendhekar A, *et al.* 1997. Aspect-oriented programming. In: Aksit M, Matsuoka S. eds. *ECOOP.* Heidelberg: Springer, 220–242.

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

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

Lopez G, Favreau R, Smith C, Costes E, Prusinkiewicz P, DeJong T. 2008. Integrating simulation of architectural development and source-sink behavior of peach trees by incorporating Markov chains and physiological organ function submodels into L-PEACH. *Functional Plant Biology* **35**: 761–771.

Minchin PEH, Snelgar WP, Blattmann P, Hall AJ. 2010. Competition between fruit and vegetative growth in Hayward kiwifruit. *New Zealand Journal of Crop and Horticultural Science* **38**: 101–112.

Perttunen J, Sievänen R. 2005. Incorporating Lindenmayer systems for architectural development in a functional–structural tree model. *Ecological Modelling* **181**: 479–491.

Pradal C, Dufour-Kowalski S, Boudon F, Fournier C, Godin C. 2008. OpenAlea: a visual programming and component-based software platform for plant modelling. *Functional Plant Biology* **35**: 751–760.

Press WH, Teukolsky SA, Vetterling WT, Flannery BP. 1992. *Numerical recipes in C: the art of scientific computing.* Cambridge: Cambridge University Press.

Prusinkiewicz P. 1986. Graphical applications of L-systems. In: Wein M, Kidd EM. eds. *Proceedings of Graphics Interface '86/Vision Interface '86.* Toronto, Canada: Canadian Information Processing Society, 247–253.

Prusinkiewicz P. 1998. Modeling of spatial structure and development of plants. *Scientia Horticulturae* **74**: 113–149.

Prusinkiewicz P. 2004. Art and science for life: designing and growing virtual plants with L-systems. *Acta Horticulturae* **630**: 15–28.

Prusinkiewicz P. 2009. Developmental computing. In: Calude CS, Costa JFGd, Dershowitz N, Freire E, Rozenberg G. eds. *Unconventional Computation. 8th International Conference, UC 2009.* Berlin: Springer, 16–23.

Prusinkiewicz P, Lindenmayer A. 1990. *The algorithmic beauty of plants.* New York: Springer.

Prusinkiewicz P, Hanan J, Měch R. 2000*a*. L-system-based plant modelling language. In: Nagl M, Schuerr A, Muench M. eds. *Applications of graph transformations with industrial relevance. Proceedings of the international workshop {AGTIVE}'99, Kerkrade, Netherlands, September 1999. Lecture Notes in Computer Science.* Berlin: Springer, 395–410.

Prusinkiewicz P, Karwowski R, Měch R, Hanan J. 2000*b*. L-studio/cpfg: a software system for modeling plants. In: Nagl M, Schurr A, Munch M. eds. *Applications of graph transformations with industrial relevance.* Berlin: Springer-Verlag, 457–464.

Prusinkiewicz P, Allen M, Escobar-Gutierrez A, DeJong T. 2007*a*. Numerical methods for transport-resistance source-sink allocation models. In: Vos J, Marcelis L, De Visser P, Struik P, Evers J. eds.

Functional–structural plant modelling in crop production. Dordrecht: Springer, 123–137.

Prusinkiewicz P, Karwowski R, Lane B. 2007*b*. The L+C plant-modelling language. In: Vos J, Marcelis L, De Visser P, Struik P, Evers J. eds. *Functional–structural plant modelling in crop production.* Dordrecht: Springer, 27–42.

Prusinkiewicz P, Crawford S, Smith RS, Ljung K, Bennett T, Ongaro V, Leyser O. 2009. Control of bud activation by an auxin transport switch. *Proceedings of the National Academy of Sciences* **106**: 17431–17436.

Sachs T. 1991. *Pattern formation in plant tissues.* Cambridge: Cambridge University Press.

Spinczyk O, Lohmann D, Urban M. 2005. AspectC++: an AOP extension for C++. *Software Developer's Journal* **5**: 68–76.

Taylor-Hell J. 2005. *Biomechanics in botanical trees.* MSc thesis, University of Calgary.

Vos J, Evers JB, Buck-Sorlin GH, Andrieu B, Chelle M, de Visser PHB. 2010. Functional–structural plant modelling: a new versatile tool in crop science. *Journal of Experimental Botany* **61**: 2101–2115.

Yokomori T. 1986. Graph-controlled systems – an extension of 0L systems. In: Rozenberg G, Salomaa A. eds. *The book of L.* Berlin: Springer-Verlag, 461–471.