

# Exploring Plant Topological Structure with the AMAPmod Software: an Outline

Christophe Godin, Evelyne Costes and Yves Caraglio

**Godin, C., Costes, E. & Caraglio, Y.** 1997. Exploring plant topological structure with the AMAPmod software: an outline. *Silva Fennica* 31(3): 357–368.

In the last decades, architectural analysis has been used to understand and to model plant development. These studies have lead us to reconsider the problem of measuring plants while taking into account their topological structure at several scales of detail. A computational platform, called AMAPmod, was created to work on such plant representations. This paper outlines the general methodology used in AMAPmod to represent plant topological structures and to explore these special types of databases. Plant structures are first encoded in order to build corresponding formal representations. Then, a dedicated language, AML, enables the user to extract various types of information from the plant databases and provides appropriate analyzing tools.

**Keywords** plant, architecture, development, coding, analysis, virtual

**Authors' addresses** *Godin & Caraglio*: CIRAD, Laboratoire de Modélisation des Plantes, BP5035, 34032, Montpellier, France. *Costes*: INRA, Laboratoire d'Arboriculture Fruitière, 2 Place Viala, 34060, Montpellier, France

**Received** 24 January 1997 **Accepted** 29 July 1997

## 1 Introduction

Topological structures of plants have been widely explored at a qualitative level since the introduction of architectural models by Hallé and Oldeman (1970). In order to take topology into account, first quantitative approaches to plant architecture made use of topology-driven sampling strategies, e.g. Honda et al. 1982, Remphey et al. 1983, Fisher and Weeks 1985, de Reffye et al. 1988, Prusinkiewicz et al. 1994,

Bouchon et al. 1997. In these approaches, samples of plant entities were designed according to topological and morphological criteria and then studied using classical statistical methods. However, this strategy assumed that the biological phenomenon of interest could be satisfactorily characterized by an *a priori* choice of topological and morphological parameters. We have now reached a second stage in which the problem is to study variations of biological phenomena as functions of their topological location. In most cases, these variations cannot be characterized

by simple direct observations. It is thus necessary to preserve the information related to plant 3D structures, including both the type of entities and their topological relationships, during the measurement processes. If topological structure is recorded and can be reconstructed within computers, then we may develop efficient computing tools for exploring these complex objects quantitatively.

We have developed such a general methodology for analyzing plants which makes use of the representation of plant topological structures. This methodology is implemented in the AMAPmod software (Godin et al. 1997). Data bases, created from field measurements of plants, can be analyzed with various exploring and modeling tools. These tools are available through a programming language named AML (AMAP Modeling Language) which enables the user to work on various objects, i.e. formal representations of plants, samples of data, or models. The user can use specialised functions to load, save, analyze, display or operate on each type of object.

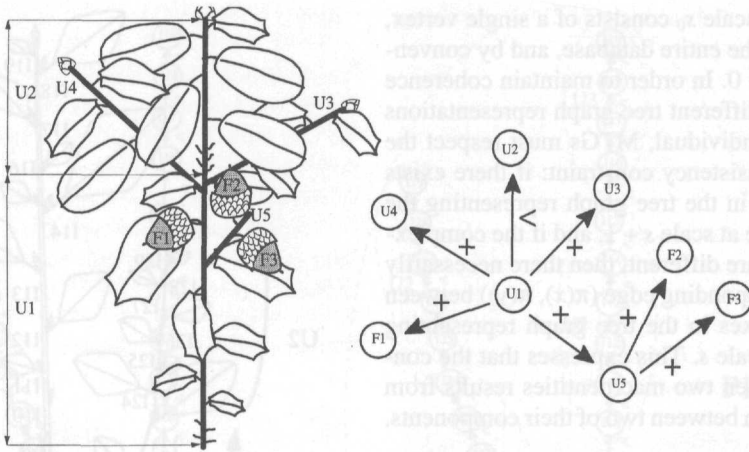
This paper gives an outline of the AMAPmod methodology for coding and exploring plants. Section 2 sketches the multiscale model of plants which has been designed in order to formally represent plant topological structures. This formal representation of plants can be encoded in textual form, which enables users to encode plant topological structures in order to process them with computers. Section 3 illustrates this coding strategy in a simple example. Section 4 shows how computer representations of plant structures can be loaded and explored within AMAPmod at several levels of complexity to illustrate different types of computing operation.

## 2 Formal Representation of Plants

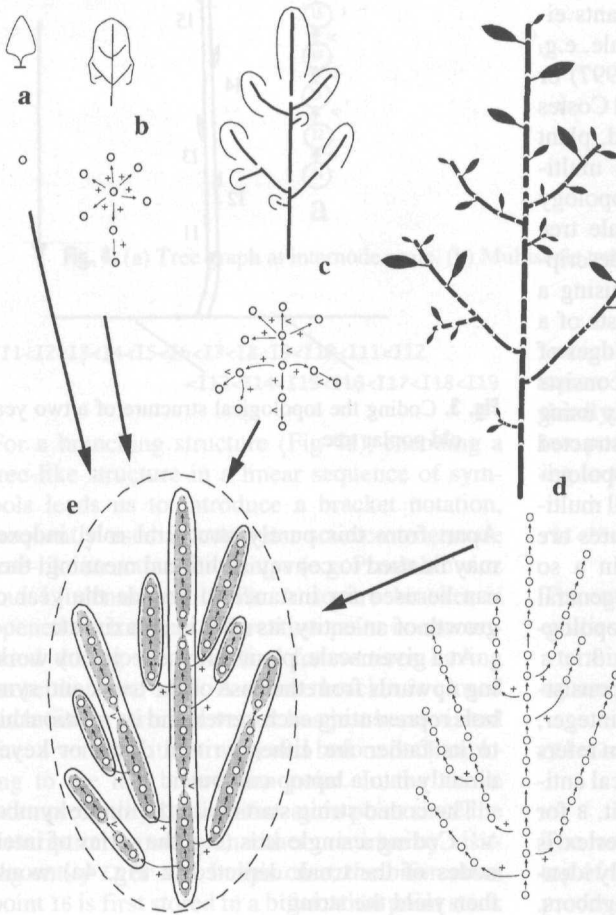
Plants are formally represented in AMAPmod by multiscale tree graphs (MTGs). The MTG formalism has been designed to enable users to express both the modularity and the multiscale nature of plant structures (Godin et al. 1997). At a given scale, plant modularity is represented by a *directed graph*. A directed graph is defined by

a set of objects, called vertices, and a binary relation between these vertices. The binary relation defines a set of ordered pairs of vertices, called edges. Vertices represent botanical entities and edges correspond to physical connections between these entities. The direction of edges respects the temporal causality of the connection between entities: edges are always directed from older entities to younger ones. Given an edge  $(a,b)$ , we say that  $a$  is a *father* of  $b$  and  $b$  is a *son* of  $a$ . Directed graphs representing plants have tree-like structures: every vertex, except the root one, has exactly one father vertex. Moreover, in order to identify the different axes of a given plant, two types of connections are distinguished: an entity can either precede (type '<') or bear (type '+') another entity. Among the sons of a given vertex, only one can be linked to its father by a '<' edge (one entity can only precede at most one entity). The other ones are necessarily linked to the father by a '+' edge. In this way, an *axis* is a maximal series of entities in the graph connected by a '<' link. In order to describe different characteristics of plant entities, vertices can have values, e.g. length, diameter, spatial location, leaf area, number of flowers, type of branched entities, etc. (Fig. 1).

A plant can be analyzed at several scales, in terms of internodes, axes or, at more macroscopic scales, in terms of branching systems. According to the above discussion, each scale of analysis corresponds to a modular structure which can be formally represented by a tree graph. In addition, we observe that entities at one scale consist of entities at a finer scale. For instance, growth units are structured sets of internodes and reiterated complexes may be analyzed as branching systems made of axes. Therefore, there exist relations between the different scales which have to be formally represented. A MTG integrates in a homogeneous framework the different tree graphs corresponding to plant descriptions at different scales (Fig. 2). Vertices at one scale are composed of vertices at a higher scale (Fig. 2e). If an entity  $a$  is composed of  $n$  entities  $x_1, x_2, \dots, x_n$ , for every  $i \in [1, n]$ ,  $a$  is called the *complex* of  $x_i$ , and  $x_i$  is a *component* of  $a$ . The complex of any entity  $x_i$  is denoted  $\pi(x_i)$ . If the scale of  $a$  is defined by the integer  $s$ , then for every  $i \in [1, n]$  the scale of  $x_i$  is  $s + 1$ . The most



**Fig. 1.** (a) Growth units constituting an annual shoot *Quercus ilex* (Fagaceae). (b) Graph representation of the growth unit organization.



**Fig. 2.** A tree at different scales of perception. The formal representation of the overall tree structure is the superposition of the partial structures existing at different scales (a) tree scale  $s = s_0 = 0$ , (b) axis scale  $s = 1$ , (c) growth unit scale,  $s = 2$ , (d) internode scale (leaves are not represented in the tree graph. They would be represented as internode attributes),  $s = 3$ , (e) corresponding multiscale tree graph.

macroscopic scale  $s_0$  consists of a single vertex, representing the entire database, and by convention has value 0. In order to maintain coherence between the different tree graph representations of the same individual, MTGs must respect the following consistency constraint: if there exists an edge  $(x,y)$  in the tree graph representing the plant structure at scale  $s + 1$ , and if the complexes of  $x$  and  $y$  are different, then there necessarily exists a corresponding edge  $(\pi(x), \pi(y))$  between these complexes in the tree graph representing the plant at scale  $s$ . This expresses that the connection between two macroentities results from the connection between two of their components.

### 3 Coding Individuals

Different strategies have been proposed for recording topological structures of real plants either for representing plant at a single scale, e.g. (Room et al. 1996, Hanan and Room 1997) or for multiscale representations (Godin and Costes 1995, Godin et al. 1997). In AMAPmod, plant topological structures are abstracted as multiscale tree graphs. Describing a plant topology thus consists of describing the multiscale tree graph corresponding to this plant. The description of a given plant can be specified using a "coding language". This language consists of a naming strategy for the vertices and the edges of multiscale graphs. A graph description consists of enumerating the vertices consecutively using their names. The name of a vertex is constructed in such a way that it clearly defines the topological location of a given vertex in the overall multiscale graph. The vertices and their features are described using this formal language in a so called "code file". Let us illustrate the general principle of this coding language by the topological structure of the plant depicted in Fig. 3.

Each vertex is associated with a label consisting of a letter, called its class, and an integer, called its index. The class of a vertex often refers to the nature of the corresponding botanical entity, e.g. I for internode, U for growth unit, B for branching system, etc. The index of a vertex is an integer which enables the user to locally identify a vertex among its immediate neighbors.

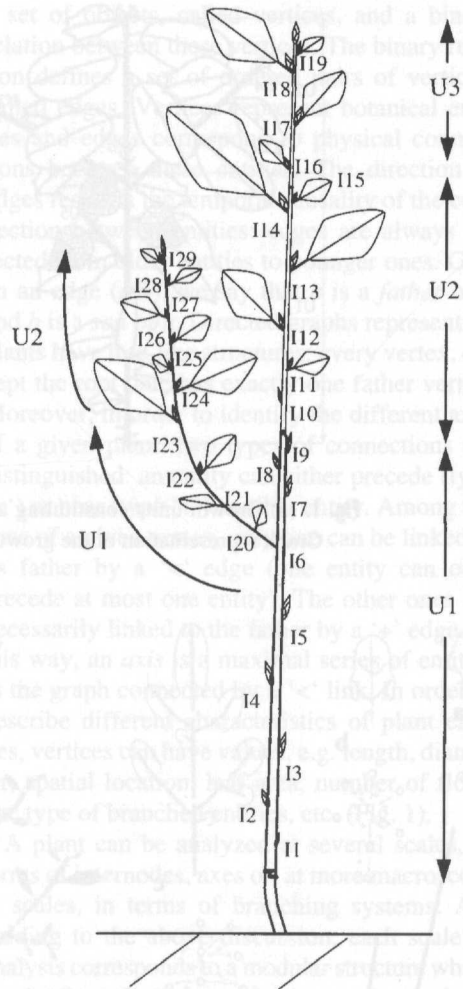


Fig. 3. Coding the topological structure of a two year old poplar tree.

Apart from this purely structural role, indexes may be used to convey additional meaning: they can be used for instance to encode the year of growth of an entity, its rank in an axis, etc.

At a given scale, plants are inspected by working upwards from the base of the trunk and symbols representing each vertex and its relationship to its father are either written down or keyed directly into a laptop computer.

The coded string starts with the single symbol '/'. Coding a single axis (e.g. the series of internodes of the trunk depicted in Fig. 4a) would then yield the string:

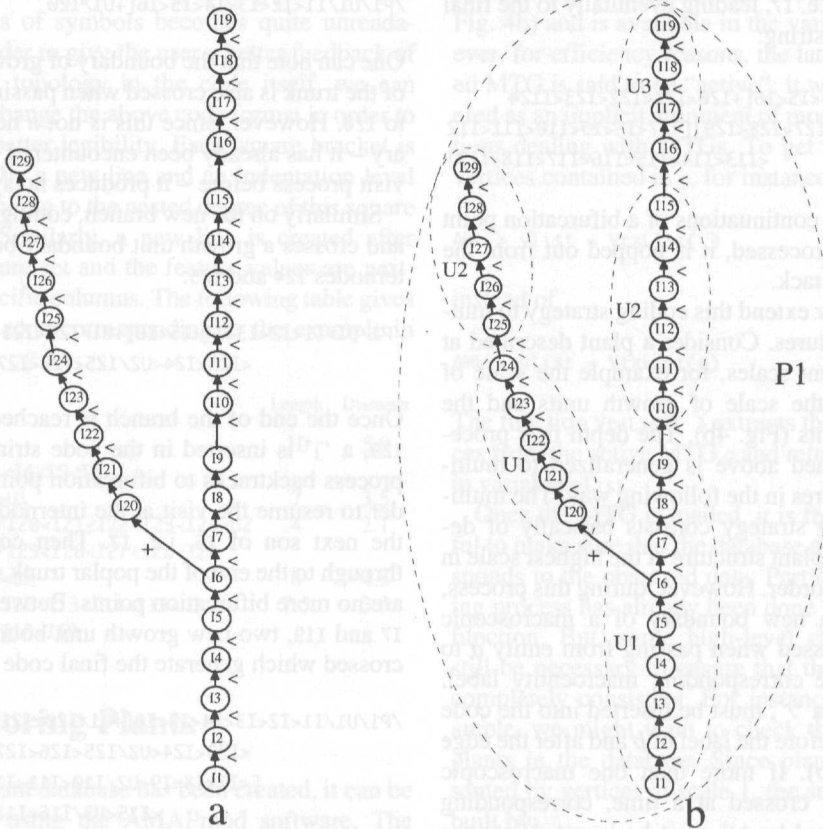


Fig. 4. (a) Tree graph at internode scale. (b) Multiscale tree graph (MTG).

```
/I1<I2<I3<I4<I5<I6<I7<I8<I9<I10<I11<I12
<I13<I14<I15<I16<I17<I18<I19
```

For a branching structure (Fig 4a), encoding a tree-like structure in a linear sequence of symbols leads us to introduce a bracket notation, frequently used in computer science to encode tree-like structures as strings (e.g. Prusinkiewicz and Lindenmayer 1990). A square bracket is opened each time a bifurcation point is encountered during the visit (i.e. for vertices having more than one son). A square bracket is closed each time a terminal vertex has just been visited (i.e. a vertex with no son) and before backtracking to the last bifurcation point. In the above example, entity I6 is a bifurcation point since the description process can either continue by visiting entity I7 or I20. In this case, the bifurcation point I6 is first stored in a bifurcation point stack

(which is initially empty). Secondly, an opened square bracket is inserted in the output string and thirdly, the visiting process resumes at one of the two possible continuations, for example I20, leading to the following code:

```
/I1<I2<I3<I4<I5<I6[+I20
```

The entire branch I20 to I28 is then encoded like entities I1 to I6. Entity I29 has no son, and thus is a terminal entity. This results in inserting a closed square bracket in the string:

```
/I1<I2<I3<I4<I5<I6[+I20<I21<I22<I23<I24
<I25<I26<I27<I28<I29]
```

The last bifurcation point can then be read at the top of the bifurcation point stack and the visiting process can resume on the next possible continu-

ation of I6, i.e. I7, leading eventually to the final output code string:

```
/I1<I2<I3<I4<I5<I6[+I20<I21<I22<I23<I24
<I25<I26<I27<I28<I29][<I7<I8<I9<I10<I11<I12
<I13<I14<I15<I16<I17<I18<I19]
```

Once all the continuations of a bifurcation point have been processed, it is popped out from the bifurcation stack.

Let us now extend this coding strategy to multiscale structures. Consider a plant described at three different scales, for example the scale of internodes, the scale of growth units and the scale of plants (Fig. 4b). The depth first procedure explained above is generalized to multiscale structures in the following way. The multiscale coding strategy consists basically of describing the plant structure at the highest scale in a depth first order. However, during this process, each time a new boundary of a macroscopic entity is crossed when passing from entity *a* to entity *b*, the corresponding macroentity label, suffixed by a '/', must be inserted into the code string just before the label of *b* and after the edge type of (*a*,*b*). If more than one macroscopic boundary is crossed at a time, corresponding labels suffixed by '/' must be inserted into the code string at the same location, labels of the most macroscopic entities first. In the multiscale graph of Fig. 4b for example, the depth first visit is carried out at the internode level (highest scale). The visit starts by entering in vertex I1 at the scale of internodes. However, to reach this entity from the outside, we cross boundaries of P1 and U1, in this order. Then the depth first visit starts by creating the code string:

```
/P1/U1/I1
```

Then, the coding proceeds through vertices I1 to I6, with no new macroscopic boundary encountered. I6 is a bifurcation point and as explained above, this vertex is stored in the bifurcation point stack, a '[' is inserted in the code string and the depth first process continues on the son of I6 whose label is I20. Since to reach I20 from I6 the new macroscopic boundary of the first growth unit of the branch is crossed, on I20 the generated code string is

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20
```

One can note that the boundary of growth unit U1 of the trunk is also crossed when passing from I6 to I20. However, since this is not a new boundary – it has already been encountered during the visit process before – it produces no symbol.

Similarly on the new branch, coding continues and crosses a growth unit boundary between internodes I24 and I25:

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20<I21<I22
<I23<I24<U2/I25<I26<I27<I28<I29]
```

Once the end of the branch is reached at entity I29, a ']' is inserted in the code string and the process backtracks to bifurcation point I6 in order to resume the visit at the internode scale on the next son of I6, i.e. I7. Then coding goes through to the end of the poplar trunk since there are no more bifurcation points. Between entities I7 and I19, two new growth unit boundaries are crossed which generate the final code string:

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20<I21<I22
<I23<I24<U2/I25<I26<I27<I28<I29]
[<I7<I8<I9<U2/I10<I11<I12<I13<I14
<I15<U3/I16<I17<I18<I19]
```

It is often the case in practical applications that a number of attributes are measured on certain plant entities. Measured values can be attached to corresponding entities using a bracket notation, '{...}'. For instance, assume that one wants to note the length and the diameter of observed growth units. For each measured growth unit, a pair of ordered values defines respectively its measured length and diameter. Then, the precedent code string would become:

```
/P1/U1{10, 5.9}/I1<I2<I3<I4<I5
<I6[+U1{7, 3.5}/I20<I21<I22<I23<I24
<U2{4, 2.1}/I25<I26<I27<I28<I29]
[<I7<I8<I9<U2{8, 4.3}/I10<I11<I12
<I13<I14<I15<U3{7.5, 3.9}/I16
<I17<I18<I19]
```

In this string, we can read that the first growth unit of the trunk, U1, has length 10 cm and diameter 5.9 mm (units are assumed to be known and fixed).

In practical applications, coding plants as raw sequences of symbols becomes quite unreadable. In order to give the user a better feedback of the plant topology in the code itself, we can slightly change the above code format in order to achieve better legibility. Each square bracket is replaced by a new line and an indentation level corresponding to the nested degree of this square bracket. Similarly, a new line is created after each feature set and the feature values are written in specific columns. The following table gives the final code corresponding to the example in Fig. 3.

	Length	Diameter
/P1/U1	10	5.9
/I1<I2<I3<I4<I5<I6		
+U1	7	3.5
/I20<I21<I22<I23<I24<U2	4	2.1
/I25<I26<I27<I28<I29		
<I7<I8<I9<U2	8	4.3
/I10<I11<I12<I13<I14<I15<U3	7.5	3.9
/I16<I17<I18<I19		

## 4 Exploring Plants

Once a plant database has been created, it can be analyzed using the AMAPmod software. The different objects, methods and models contained in AMAPmod can be accessed through a functional language called AML. This language has been designed to optimize access to plant databases.

### 4.1 Creating Plant Representations

The formal representation of a plant, and more generally of a set of plants, can be built by AMAPmod from its code file using the AML function `MTG( )`:

```
AML > g = MTG("tree_code_file.txt")
```

The procedure `MTG` attempts to build the plant formal representation, checking for syntactic and semantic correctness of the code file. If the file is not consistent, the procedure outputs a set of errors which have to be corrected before applying a new syntactic analysis. Once the file is

syntactically consistent, the MTG is built (cf. Fig. 4b) and is available in the variable `g`. However, for efficiency reasons, the latest constructed MTG is said to be "active": it will be considered as an implicit argument of most of the functions dealing with MTGs. To get the list of all vertices contained in `g`, for instance, we write:

```
AML > vlist = VtxList( )
```

instead of

```
AML > vlist = VtxList(g)
```

The function `VtxList( )` extracts the set of vertices from the active MTG `g` and returns the result in variable `vlist`.

Once the MTG is loaded, it is frequently useful to make sure that the database actually corresponds to the observed data. Part of this checking process has already been done by the `MTG( )` function. But, some high-level checking may still be necessary to ensure that the database is completely consistent. For instance, in our example, we might want to check the number of plants in the database. Since plants are represented by vertices at scale 1, the set of plants is built by:

```
AML > plants = VtxList(Scale → 1)
```

Like `vlist`, the set `plants` is a set of vertices. The number of plants can be obtained by computing the size of the set `plants`.

```
AML > plant_nb = Size(plants)
```

Each plant constituting the database can be individually and interactively accessed via AML. For instance, assuming the plant corresponding to the example of Fig. 4b is represented by a vertex (at scale 1) with label P1. Plant P1 can be identified in the database by selecting the vertex at scale 1 having index 1:

```
AML > plant1 = Foreach _p In plants:
Select(_p, Index(_p)==1)
```

In this expression, the `Foreach` construct is used to browse the set of plant vertices `plants`. For

each plant vertex  $_p$  in this set, operator `Select` is applied and returns a non void value only for vertices whose index value is 1. `Plant1` thus contains the vertex representing plant P1. Now it is possible to apply new functions to this vertex in order to explore the nature of plant P1. Assume for instance we want to know the number of growth units composing P1:

```
AML > gu_nb = Size(Components(plant1))
```

`Components()` is a built-in function which applies to a vertex  $v$  and returns the vertices composing  $v$  at the next superior scale. Since `plant1` is a vertex at scale 1, representing plant P1, components of `plant1` are vertices at scale 2, i.e. growth units. It is also possible to compute the number of internodes composing a plant by simply specifying the optional argument `Scale` in function `Components`:

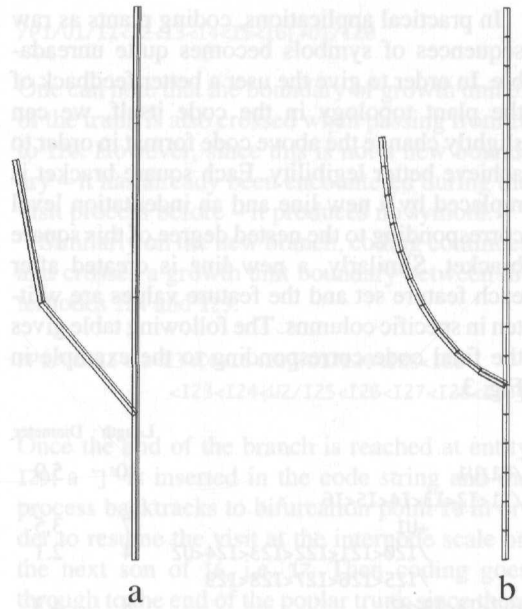
```
AML > internode_nb =  
      Size(Components(plant1, Scale → 3))
```

Many such direct queries can be made on the plant database which provide interactive access to it. However, a complementary synthesizing view of the database may be obtained by a graphical reconstruction of plant geometry. Geometrical parameters, like branching and phyllotactic angles, diameters, length, shapes, are read from the database. If they are not available, mean values can be inferred from samples or from additional data describing plant general geometry (Godin et al. 1996). A 3D interpretation of the MTG provides the user with natural feedback on the database. Built-in function `PlantFrame()` computes the 3D-geometry of plants. For example,

```
AML > frame1 = PlantFrame(plant1)
```

computes a 3D-geometrical interpretation of P1 topology at scale 2, i.e. in terms of growth units (Fig. 5a). Like in the previous example, `PlantFrame()` takes `Scale` as an optional argument which enables us to build the 3D-geometrical interpretation of P1 at the level of internodes (Fig. 5b):

```
AML > frame2 = PlantFrame(plant1, Scale → 3)
```



**Fig. 5.** 3D geometrical reconstruction of the MTG. Reconstruction (a) at growth unit scale. (b) at internode scale.

Refinements of this 3D geometrical reconstruction may be obtained with the possibility to change the shape of the different plant components, possibly at different scales, to tune geometrical features (length, diameter, insertion angle, phyllotaxy, ...) as functions of the topological position of entities in the plant structure.

## 4.2 Extraction of Plant Entity Features

When attributes of entities are available in MTGs, it is possible to retrieve their values by using the function `Feature()`:

```
AML > first_gu = Trunk(plant1)@1
```

```
AML > first_gu_diameter =  
      Feature(first_gu, "Diameter")
```

The first line retrieves the vertex corresponding to the first growth unit of the trunk of P1 (function `Trunk()` returns the ordered set of components of the trunk of vertex P1, and operator `@` with argument 1 selects the first element of this

set). Then, in the second line, the diameter of this growth unit is extracted from the database. Variable `first_gu_diameter` then contains the value 5.9 (see the code file). Similarly the length of the first growth unit can be retrieved:

```
AML > first_gu_length =
      Feature(first_gu, "Length")
```

Variable `first_gu_length` contains value 10.

The user can simplify this extraction by creating alias names:

```
AML > diameter(_x) = Feature(_x, "Diameter")
```

```
AML > length(_x) = Feature(_x, "Length")
```

It is then possible with these functions to build data arrays corresponding to feature values associated with growth units.

```
AML > growth_unit_set = VtxList(Scale → 2)
```

```
AML > Foreach _x In growth_unit_set: length(_x)
```

Moreover, new synthesized attributes can be defined by creating new functions using these basic features. For example, making the simple assumption that the general form of a growth unit is a cylinder, we can compute the volume of a growth unit:

```
AML > volume(_x) =
      (PI*diameter(_x)^2 / 4)*length(_x)
```

where  $\pi$  denotes the real constant  $\pi$  and  $^{\wedge}$  denotes the power function. Now, the user can use this new function on any growth unit entity as if it were a feature recorded in the MTG. For instance, the volume of the first growth unit is computed by:

```
AML > first_gu_length = volume(first_gu)
```

The total volume of the trunk:

```
AML > trunk_volume = Sum( Foreach _gu
      In Trunk(plant1) : volume(_gu) )
```

The wood volume of the whole plant can be computed by:

```
AML > plant_volume = Sum( Foreach _gu
      In Components(plant1) : volume(_gu) )
```

### 4.3 Extracting More Information from Plant Databases

As illustrated in the previous section, plant databases can be investigated by building appropriate AML queries. Built-in words of the AML language may be combined in various ways in order to create new queries. In this way, more and more elaborated types of queries can be constructed by creating user-defined functions which are equivalent to computing programs. In order to illustrate this procedure, let us assume that we would like to study distributions of numbers of internodes per growth units, such distributions being an important basic prerequisite for botanically-based 3D plant simulations (e.g. Barthélémy 1991, de Reffye et al. 1991, Jaeger and de Reffye 1992, Bouchon et al. 1997). At a first stage, we consider all the growth units contained in the plant database together. We first need to define a function which returns the number of internodes of a given growth unit. Since in the database, each growth unit (at scale 2) is composed of internodes (at scale 3) we compute the set of internodes constituting a given growth unit `_x` as follows:

```
AML > internode_set(_x) = Components(_x)
```

The object returned by function `internode_set( )` is a set of vertices. The number of internodes of a given growth unit is thus the size of this set:

```
AML > internode_nb(_x) =
      Size(internode_set(_x))
```

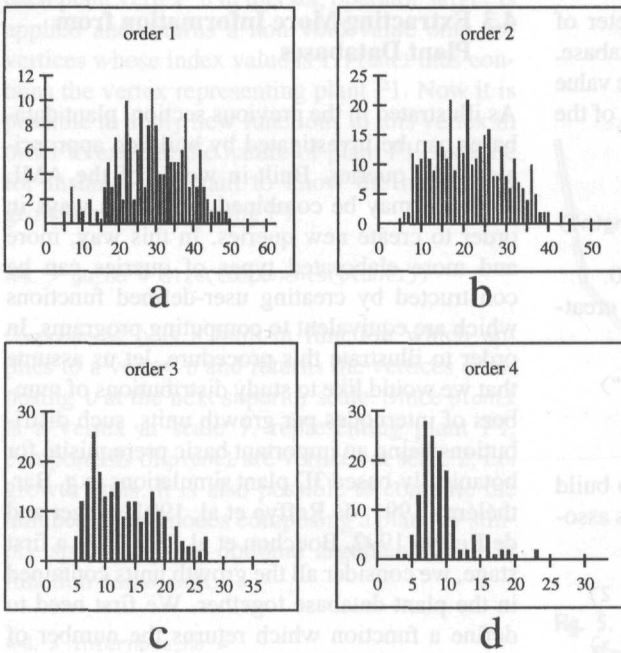
Second, the entities on which the previous function has to be applied, must be located in the database. A set of vertices is created by selecting plant entities having a certain property.

The set of growth units is the set of entities at scale 2:

```
AML > gu_set = VtxList(Scale → 2)
```

Third, we have to apply function `internode_nb( )` to each element of the selected set of entities:

```
AML > sample1 =
      Foreach _x In gu_set : internode_nb(_x)
```



**Fig. 6.** Different distributions of the number of internodes per growth unit, in different topological situations.

We use iterator `Foreach` in order to browse the whole set of growth units of the database, and to apply our `internode_nb()` function to each of them.

Now, we want to get the distribution of the number of internodes on a more restricted set of growth units. More precisely, we would like to study the distribution of internode numbers of different populations corresponding to particular locations in the plant structure. We thus have to define these populations first and then to iterate the function `internode_nb()` on each entity of this new population like in the previous example. Let us consider for example the population made of the growth units composing branches of order 1. Consider again the whole set of growth units `gu_set`. Among them, those which are located on branches (defined as entities of order 1 in AML) are defined by:

```
AML > gu1 = Foreach _x In VtxList
  (Scale -> 2) : Select(_x, Order(_x) == 1)
```

Here again, we use the iterator `Foreach` in order to browse the whole set of growth units of the database, and to apply the `Select` operator to

each of them. `Select` will return only growth unit vertices whose order is 1. AML variable `gu1` thus contains all the growth units in the corpus which are located on branches. Eventually, after the sample of values is built, the above function is applied to the selected entities:

```
AML > sample = Foreach _x
  In gu1 : internode_number(_x)
```

At this stage, a set of values has been extracted from the plant database corresponding to a topologically selected set of entities. This sample of data can be further investigated with appropriate AML tools. For example, AML provides the built-in function `Histogram()` which builds the histogram corresponding to a set of values.

```
AML > histo1 = Histogram(sample)
```

```
AML > Plot(histo1)
```

This plot gives the graph depicted in Fig. 6a. Similarly, by selecting samples corresponding to different topological situations, we would obtain the series of plots in Fig. 6 (Caraglio et al. 1990).

## 5 Conclusion

This paper outlined the general methodology for representing, encoding and exploring plants as defined in AMAPmod, by giving simple and fully explained examples. Using the AML language, the user can define complex queries by combining functions and design in this way his own exploring strategy. Since plants are represented in databases by structured objects, i.e. MTGs, queries may return objects which preserve parts of the structure of MTGs: e.g. spatial or temporal sequences of events or tree-structured data. Tools for studying these complex objects are currently under development in AMAPmod (Godin et al. 1997, Guédon et al. 1995, Costes and Guédon 1997).

Moreover, preservation of plant topological and spatial information in measurements enables us to achieve better independence between plant databases and end-user applications: potentially, various types of analyses can be applied on the same database. In addition to studying morphological characteristics (architecture, geometry, form, allometric relations, ...), plant structure descriptions can be used to study the coupling of structure and physiological functions (photosynthesis, fruiting, mechanical support, water transport, etc.).

The will to have a common language which can be used to describe a wide range of species and the need to factorize the development of computer tools to work on plant structure databases have been major motivations in the development of AMAPmod. As a consequence, a current key issue in using AMAPmod is related to the definition of plant corpora. This would firstly enable research scientists to efficiently exchange plant data. Secondly, this would enable modelers to compare their models on the basis of public (or at least partially shared) sets of data. As a counterpart, data collection may be more expensive in this more general perspective than in the context of a precise study. This might not be a too severe limitation if the definition of plant corpora are justified in terms of facilitating collaborative research.

## References

- Barthélémy, D. 1991. Levels of organization and repetition phenomena in seeds plants. *Acta Biotheoretica* 39: 309–323.
- Bouchon, J., de Reffye, P. & Barthélémy, D. (eds.) 1997. *Modélisation et simulation de l'architecture des végétaux*. Science Update. INRA éditions.
- Caraglio, Y., Elguero, E., Mialet, I. & Rey, H. 1990. *Le Peuplier. modélisation et simulation de son architecture*. Rapport annuel, convention idf/cirad, CIRAD, Laboratoire de Modélisation des Plantes.
- Costes, E. & Guédon, Y. 1997. Modelling the sylleptic branching on one-year-old trunks of apple cultivars. *Journal of American Society for Horticultural Science* 122(1): 53–62.
- de Reffye, P., Dinouard, P. & Barthélémy, D. 1991. *Modélisation et simulation de l'architecture de l'Orme du Japon Zelkova serrata Thunb. Makino Ulmaceae: la notion d'axe de référence*. In: *L'Arbre, Biologie et développement*. *Naturalia Monspelienis*: 251–266.
- Edelin, C., Françon, J., Jaeger, M. & Puech, C. 1988. Plant models faithful to botanical structure and development. In: Dill, J. (ed.), *Proceedings of SIGGRAPH'88*, vol. 22, p. 151–158, Atlanta. Computer Graphics.
- Fisher, J. & Weeks, C. 1985. Tree architecture of *Neea* (Nyctaginaceae): geometry and simulation of branches and the presence of two different models. *Bull. Mus. natn. Hist. nat* 7: 385–401.
- Godin, C., Bellouti, S. & Costes, E. 1996. Restitution virtuelle de plantes réelles: un nouvel outil pour l'aide à l'analyse de données botaniques et agronomiques. In: *Proceedings of the Interface to Real and Virtual Worlds*, p. 369–378.
- & Costes, E. 1995. How to get representation of real plants in computers for exploring their botanical organization. In: *Proceedings of the 4th International Symposium on Computer Modelling in Fruit Research and Orchard Management*, p. 45–52.
- , Guédon, Y., Costes, E. & Caraglio, Y. 1997. Measuring and analyzing plants with the AMAPmod software. In: Michalewicz, M. (ed.), *Advances in computational life sciences I: Plants to ecosystems*, p. 63–94. CSIRO, Australia.
- Guédon, Y., Costes, E. & Caraglio, Y. 1995. *Modélisation de structures végétales résultant de la suc-*

