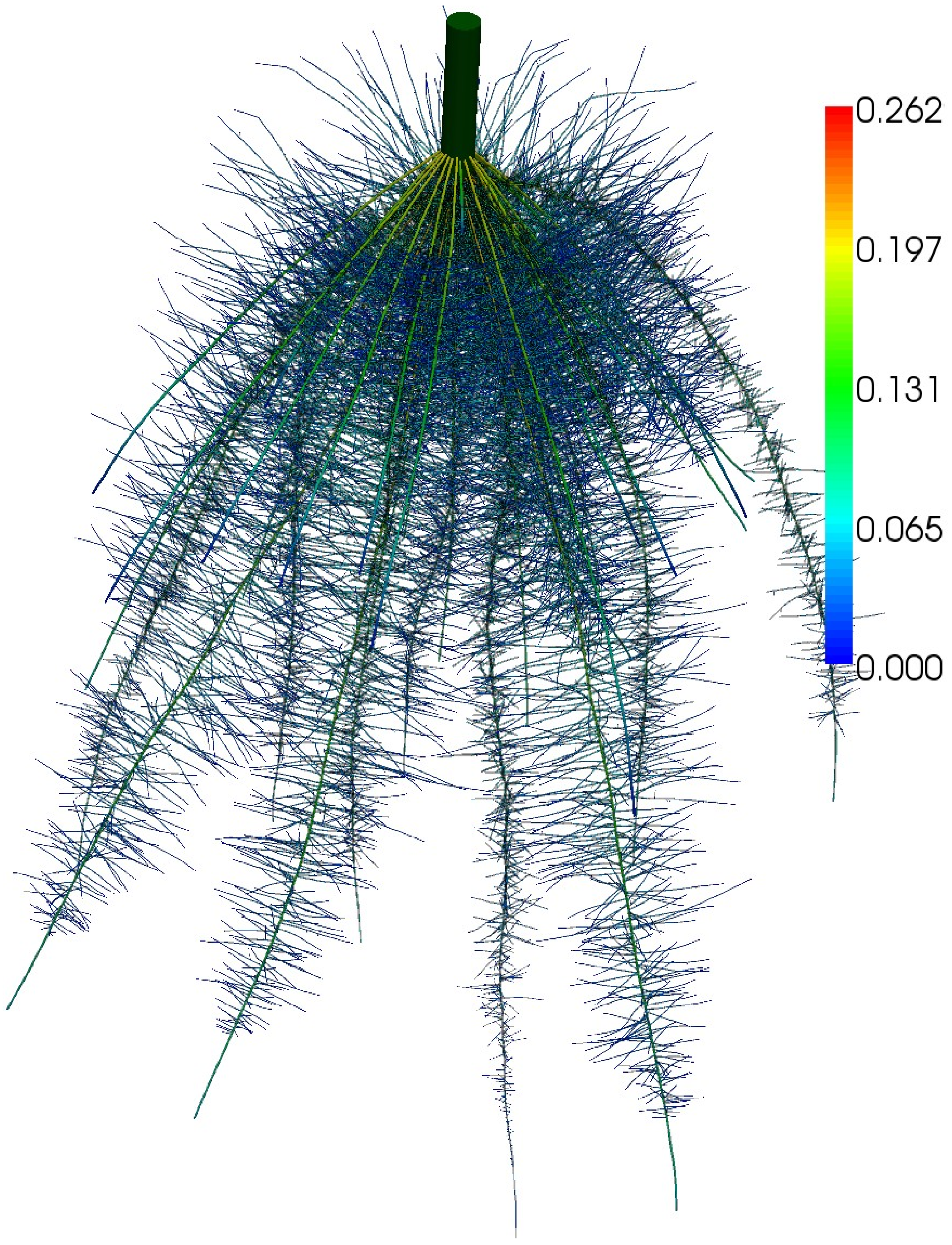
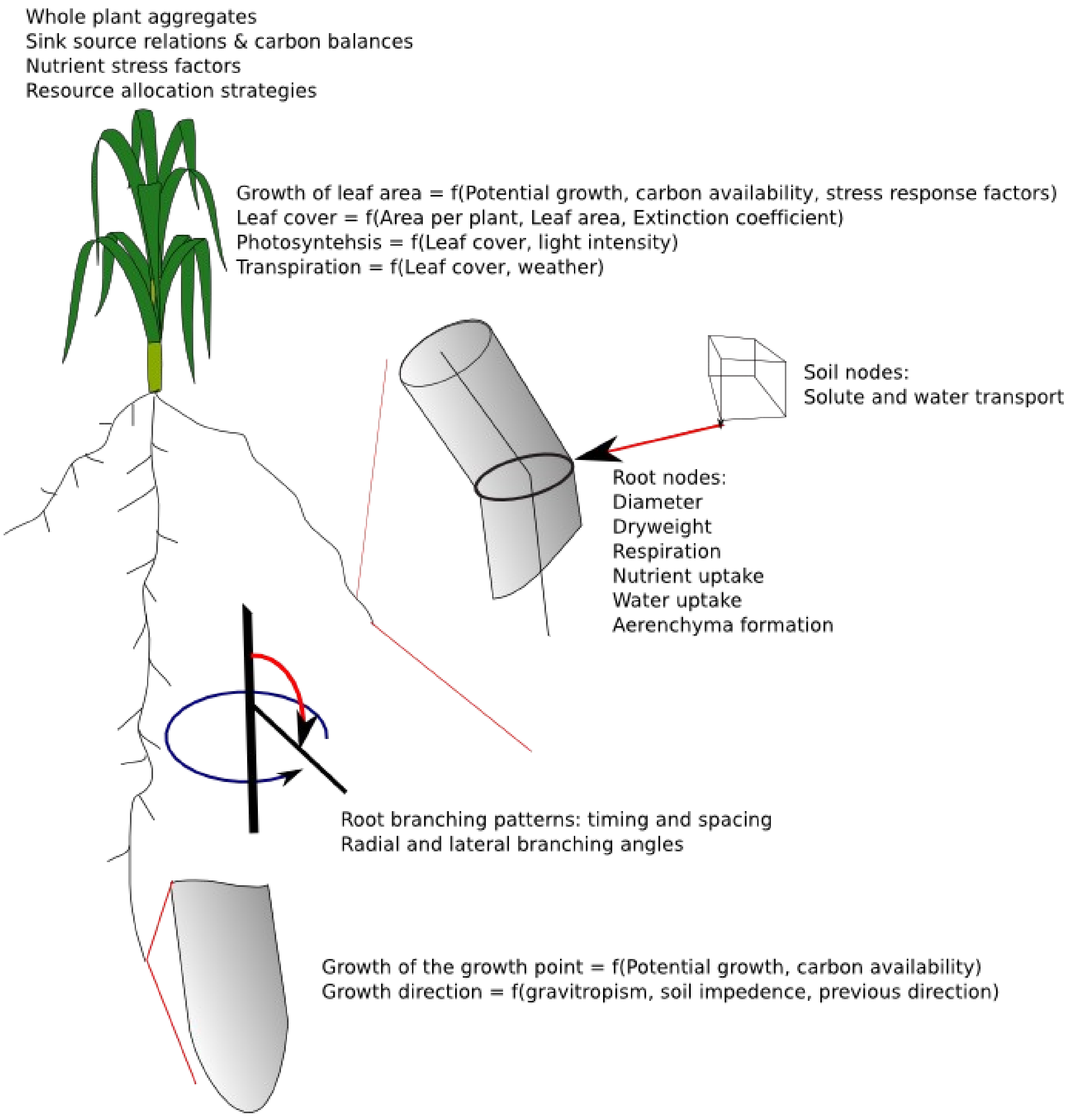
SimRoot Manual



Introduction

SimRoot is a 3D architectural root model. Version 4 and earlier were written in c. This manual only is only useful for the new SimRoot which is a complete rewrite. The new SimRoot is modular and adaptive and thereby more useful, but possibly more complicated to learn. This manual should help you on the way.

Figure 1 Global presentation of different components of SimRoot

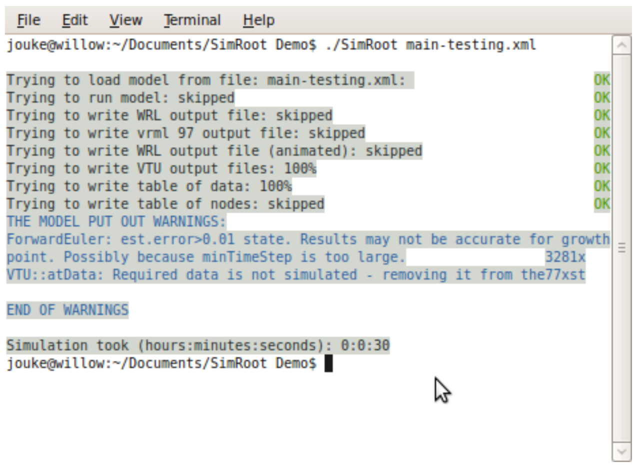

Whole plant aggregates
Sink source relations & carbon balances
Nutrient stress factors
Resource allocation strategies
Growth of leaf area = f(Potential growth, carbon availability, stress response factors)
Leaf cover = f(Area per plant, Leaf area, Extinction coefficient)
Photosyntehsis = f(Leaf cover, light intensity)
Transpiration = f(Leaf cover, weather)
Soil nodes:
Solute and water transport
Root nodes:
Diameter
Dryweight
Respiration
Nutrient uptake
Water uptake
Aerenchyma formation
Root branching patterns: timing and spacing
Radial and lateral branching angles
Growth of the growth point = f(Potential growth, carbon availability)
Growth direction = f(gravitropism, soil impedence, previous direction)

Usage

SimRoot has currently a command line user inteface (CLI). So you will have to run it in a terminal. If you have never used a terminal, is good to learn basic commands like cd and ls. Since SimRoot uses 'ANSI/VT100 Terminal Control Escape Sequences' for formatting it's output, you are best of using a terminal that supports it (otherwise you may see a lot of gibberish between the output). I use bash. Running SimRoot without arguments will give you a small help output. Basically you should run it with the '-f filename' option where filename is you simulation file name. All other options are controlled from the input file(s). For info on the input file see further.

When you run SimRoot it will show you progress. You can stop the run with ctrl+c key combination. This will pause the execution and ask you if you want to terminate the run or if you just want to see the warning messages. Answering y will terminate the execution, n will print the warnings and continue.

Figure 2 SimRoot finishing a run in the terminal. A set of warning messages in blue.



```
jouke@willow:~/Documents/SimRoot Demo$ ./SimRoot main-testing.xml

Trying to load model from file: main-testing.xml:                         OK
Trying to run model: skipped                                              OK
Trying to write WRL output file: skipped                                  OK
Trying to write vrml 97 output file: skipped                              OK
Trying to write WRL output file (animated): skipped                       OK
Trying to write VTU output files: 100%                                    OK
Trying to write table of data: 100%                                       OK
Trying to write table of nodes: skipped                                   OK
THE MODEL PUT OUT WARNINGS:
ForwardEuler: est.error>0.01 state. Results may not be accurate for growth
point. Possibly because minTimeStep is too large.                    3281x
VTU::atData: Required data is not simulated - removing it from the77xst

END OF WARNINGS

Simulation took (hours:minutes:seconds): 0:0:30
jouke@willow:~/Documents/SimRoot Demo$ 
```

SimRoot's output and error messages

SimRoot's output depends on your setting in the inputfiles. You'll find these settings in the 'simulationControlParameters' section. The main output are a custom table and vtk output for visualisation with paraview.

I will not describe paraview here, you can find tutorials for that online, but I do want to give you some tips on the tabled output. This tabled output is simply a text file with a fixed set of columns. These colums are the name of the parameter, time the value of the parameter, the unit and the context of that parameter in the database. The file can be very long. To quickly find what I need I use a set of command line utilities: grep, cut and quickplot. Grep filters lines, cut extracts colums and quickplot makes a quick plot. You can combine these command with a pipe (|) which connect the output of one program to the input of another. Here is an example:

```
cat tabled_output.tab | grep rootLength | grep -v plantPosition | cut -f2,3 | quickplot
```

This examples reads the table, filters out the lines that contains the word rootLength, removes lines that contain the word plantPosition (so I only get the rootLength of the whole plant, and not of the branches), cuts out column 2 and 3 which are the time and the value, and makes a plot of time against value. With a little creativity you can come up with simple queries that will work for you. If you rather had data in a spreadsheet, you can do something like:

```
cat tabled_output.tab | grep rootLength | grep -v plantPosition > mydatasubset.dat
```

This would write a file mydatasubset.dat with the filtered data in it. Now you can open that file with a spreadsheet. A quick listing on the screen is possible with something like:

```
cat tabled_output.tab | grep rootLength | grep -v plantPosition | less
```

use q to get out of less. I encourage you look at the manual pages of the different commands to learn more about how you can use the text utilities to do automated processing of data. I use shell scripts to generate reports that are usefull for a specif study. Thus I can easily combine output from different simulation runs and compare them without doing much work.

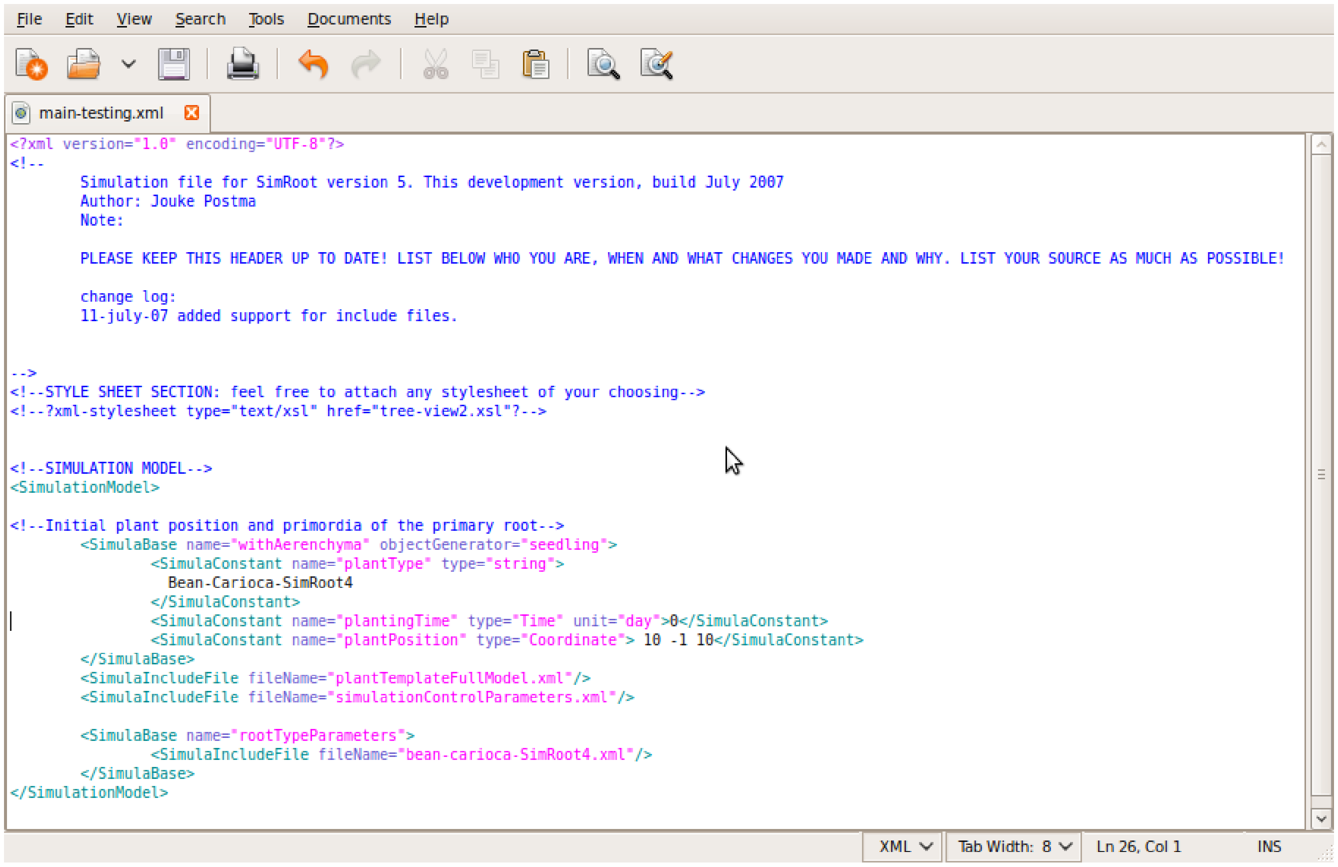
SimRoot makes a distinction between errors and warnings. Errors will terminate execution while warning will be listed on the screen at the end of the execution. Most errors will indicate where the error occurred and a hint about what went wrong. Generally an error means you'll have to change something in the input file or the code. If it is an error in the code and need further analysis – a debugger (GDB with some graphical front end) may be your best tool.

The input files

SimRoot input files are more extensive than most models. The input files not just a list of parameters, but describe your whole simulation system, including all the modules that you will use for the different processes. The input files are hierarchical. The hierarchy gives context to the different parameters and modules. For examples, root parameters are part of a root class, which is part of a genotype.

The input files are formatted according to the xml rules. If you not familiar with xml you'll have to look it up on the web. The input files are normal text files (as opposed to a binary format) – so you are best off using an xml aware text editor so you have nice color coding and checking of xml rules. I use gedit. I haven't written a schema or DDT (yet) – but SimRoot is pretty good at telling you if there is something wrong with your formatting.

Figure 3 Example input file opened in gedit


File Edit View Search Tools Documents Help
main-testing.xml
XML Tab Width: 8 Ln 26, Col 1 INS

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
	Simulation file for SimRoot version 5. This development version, build July 2007
	Author: Jouke Postma
	Note:

	PLEASE KEEP THIS HEADER UP TO DATE! LIST BELOW WHO YOU ARE, WHEN AND WHAT CHANGES YOU MADE AND WHY. LIST YOUR SOURCE AS MUCH AS POSSIBLE!

	change log:
	11-july-07 added support for include files.


-->
<!--STYLE SHEET SECTION: feel free to attach any stylesheet of your choosing-->
<!--?xml-stylesheet type="text/xsl" href="tree-view2.xsl"?-->


<!--SIMULATION MODEL-->
<SimulationModel>

<!--Initial plant position and primordia of the primary root-->
	<SimulaBase name="withAerenchyma" objectGenerator="seedling">
		<SimulaConstant name="plantType" type="string">
		  Bean-Carioca-SimRoot4
		</SimulaConstant>
		<SimulaConstant name="plantingTime" type="Time" unit="day">0</SimulaConstant>
		<SimulaConstant name="plantPosition" type="Coordinate"> 10 -1 10</SimulaConstant>
	</SimulaBase>
	<SimulaIncludeFile fileName="plantTemplateFullModel.xml"/>
	<SimulaIncludeFile fileName="simulationControlParameters.xml"/>

	<SimulaBase name="rootTypeParameters">
		<SimulaIncludeFile fileName="bean-carioca-SimRoot4.xml"/>
	</SimulaBase>
</SimulationModel>
```

The main file structure

The main simulation file should open with the '<SimulationModel>' tag. Between the opening and closing tag you can define your model and model parameters. There are sections for:

1) the simulation control parameters
2) the plant and environmental parameters
3) the different plant, shoot, hypocotyl and root templates

Finally you should include 1 or more plants and optionally a soil section.

The easiest way is to take an existing (set of) file(s) and modified those to your needs. The files should be largely self explanatory.

Include files: splitting the main file up.

For your convenience SimRoot allows you to reference parts of your model in include files which should start with the '<SimulationModelIncludeFile>' tag. These files should be referenced inside the main file with a '<SimulaIncludeFile fileName="putHereTheNameOfYourFile.xml"/>. The rules for the included files are identical to those of the main file. For example you can include an include file inside another include file.

The tree like structure of the xml file should be preserved when you split the input of several files. Names of the simula objects that you reference in the input file should be unique within that context i.e. two objects that have the same parent should have different names. Therefor, you can not declare the same object twice in different input files. However – since it is handy to insert certain objects into the tree in a different input file (which you may include for one simulation but not another one) you can use a '<SimulaDirective path="your path">' tag to insert the children of this tag somewhere else. Note that the path should be a list of tags that has already been read into the database. So the order of reading matters here (Yes – this may violate the xml rules slightly). SimRoot reads the input files sequentially. The path is specified as you would specify a directory path on the command line. Both relative and absolute path's are supported. A little example may help.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!--
put some comment here about your what project this belongs too, SimRoot version etc.
-->

<SimulationModel>
       <SimulaIncludeFile fileName="simulationControlParameters.xml"/>
       <SimulaIncludeFile fileName="myplants.xml"/>
       <SimulaIncludeFile fileName="environmentalParameters.xml"/>
       <SimulaIncludeFile fileName="plantTemplate.xml"/>
       <SimulaBase name="rootTypeParameters" />
       <SimulaIncludeFile fileName="plantParameters/maize.xml"/>
       <SimulaIncludeFile fileName="plantParameters/bean.xml"/>
       <SimulaIncludeFile fileName="plantParameters/squash.xml"/>
</SimulationModel>
```

Example of a minimal input file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SimulationModelIncludeFile>
       <SimulaDirective path="/rootTypeParameters">
              <SimulaBase name="maize">
              ...
              ...
              ...
              ...
              </SimulaBase>
       </SimulaDirective>
</SimulationModelIncludeFile>
```

Example of a minimal include file

What tags does SimRoot understand?

There is a limited number of tags that SimRoot understands. These are:

1) SimulationModel
2) SimulationModelIncludeFile
3) SimulaDirective
4) SimulaBase
5) SimulaConstant
6) SimulaStochastic
7) SimulaTable
8) SimulaDerivative
9) SimulaVariable
10) SimulaPoint

Of these the first three have been explained before. The others are used to create simulation objects in the database. Each of tags can have an unlimited number of other SimulaX tags included thus building a tree of simulation objects. This tree is identical to the structure of the database in SimRoot.

Tags that simulate the same data type, most likely a real number, are completely exchangeable. For example, a parameter that is constant in one simulation can be defined as an interpolation table or an stochastic variable in another simulation. This does not require any changes to SimRoot's code.

Each SimulaX object has a name attribute which should be a unique name among the objects that have the same parent. The name is usually self explanatory and is specified with a name="tagsName" attribute.

SimulaBase

SimulaBase objects are used to group objects together under a name. SimulaBase objects can have optionally the 'objectGenerator' attribute set. If set – the specified object generator will create new simulation objects as time progresses. For example for each change in direction of the growth point a new datapoint is created, and has a root growth, more branches are generated.

SimulaConstant

SimulaConstant represents the most simple simulation object, a constant. This object can take different data types. Currently real numbers, integers, strings and boolean are supported. You have to set the type attribute or otherwise the program will assume you are simulating a real number. The type attribute takes as value: numeric, State, state; Time, time; integer, int; text, string; Coordinate, coordinate, Point, point; Bool, bool. You can specify the units of the parameter if needed. Many modules do check the units, so you are best of to be explicit. Here are some examples:

```
<SimulaConstant name="run" type="bool"> true </SimulaConstant>
<SimulaConstant name="growthRate" unit=”cm/day”> 4.0 </SimulaConstant>
<SimulaConstant name="plantType" type=”string”> maize </SimulaConstant>
```

SimulaStochastic

SimulaStochastic returns a value from a distribution. Both integer and real numbers are supported. The distribution attribute can be uniform, normal or lognormal. For simulation of uniform distribution you should specify a minimum and maximum value – while for the (log)normal distributions you have to supply a mean and stdev.

```
<SimulaStochastic type="integer" name="myrandominteger" unit="#" distribution="uniform"
minimum="0" maximum="6" />
<SimulaStochastic type="double" name="geotropism" unit="#" distribution="normal"
mean="0.01" stdev="0.001" />
```

SimulaTable

SimulaTable does linear interpolation (other interpolation methods may be implemented in the future). It takes two columns of data and returns the interpolated value. You have to specify the name of both columns and the units of both columns. There can be an unlimited number of rows and two columns (support for more dimensional data is not implemented yet). The rows will be automatically sorted on the first column. Duplicate values will be overwritten without warning with the last listed value the one that will be used. Note that extrapolation will result in a simulation error. You'll have the provide data for the appropriate range.

```
<SimulaTable name_colum1="time since creation" unit_colum1="day"
name_colum2="relativeRespiration" unit_colum2="g/g/day">
 0     0.09
 2     0.09
 6     0.04
 1000  0.04
</SimulaTable>
```

SimulaDerivative

SimulaDerivative uses one of the plugin modules from the library of scientific modules to simulate data. You'll have to supply the name of the module and the the type of data the module returns. If the type attribute is omitted it is assumed the module returns real numbers.

```
<SimulaDerivative name="rootLength" unit="cm" function="plantTotalRootFraction" />
```

To reduce computational time – SimulaDerivative interpolates between previous calculated times when the estimated interpolation error is small. However when preferedTimeStep is set – it will interpolate when the time between two datapoints is less than the preferred timestep. Similar – minTimeStep will be honored.

SimulaVariable

SimulaVariable does dynamic simulation. It takes an initial state as input and calculates the rate of change of the state using a user specified function. Than it integrates. The default integration method is RungeKutta 4, but you can use Forward Euler or Heuns by specifying it with the integrationFunction attribute. SimulaVariable requires that the module that is used for the rate calculations is specified with the rateFunction attribute (you may write "function" for short).

The time-stepping module that is used can be controlled with a number of optional arguments: the minimum and maximum time step and the preferred time step. If the preferred time step is larger than the minimum time step, the preferred time step will be used instead of a for (estimated) error adjusted time step. Currently, I am experimenting with the possibility to have synchronized time steps between modules. This is done by using the minimum time step as a round of factor.

```
<SimulaVariable name="weight" unit="g" rateFunction="useDerivative" integrationFunction=
"euler" preferedTimeStep="0" maxTimeStep="1." minTimeStep="0.001">0</SimulaVariable>
```

SimulaPoint

SimulaPoint simulates a moving point in 3d space. The unit for the coordinate system is in cm. It takes an initial position as input and uses a specified function to simulate the direction and speed of the movement. It integrates in all 3 dimensions to determine the new position. The integration and time-stepping modules behaves the same as explained under SimulaVariable. However, the timestepping module can also adjust the timestep such that the default change in position is always a specified distance from the previous position. This is done by specifying a non zero "defaultIntegrationLength" in the "integrationParmeters" section under "simulationControls" (See further for a description of the different sections in the input file). All SimulaPoint objects will honor this setting. It is especially useful for generating more smooth images, but may slow down the simulation dramatically when a small value is chosen.

```
<SimulaPoint name="growthpoint" rateFunction="rootgrowth" integrationFunction="euler"
preferedTimeStep="0" maxTimeStep="1." minTimeStep="0.001">
	0 0 0 <!--initial x,y,z location in cm, positive y is up-->
</SimulaPoint>
```

Major sections in the input file

Simulation control parameters

The simulation control parameter section will tell SimRoot what export modules to run. Each export module has it's own section under it's own name. For each export module you can define whether it should be run or not – and at what time it should start the output, finish the output and what timestep to use for the output. For example you may want vtk output ten times a day, while for the tabled output you may only want daily data. Some export modules look for optional parameters. For example tabled output looks for a list of parameters it should include in it's table. You will have to check the documentation / code of the modules that you are using to know what these parameters are, or you can just look at an example input file.

There is currently one (optional) simula-control parameter that is *not* associated with an export module. It is the default spatial integration length. When set to 0 or omitted – SimRoot will try to estimate the time step for moving the growthpoints of the roots based on a simple error rule which will try to keep the numerical error in the position of the growthpoint less than 1% of the distance the growthpoint moved. For smooth images you may rather want the growthpoint to move 1 mm at a time. So you can set this parameter to 0.1 cm. (see also under SimulaPoint).

```
<SimulaBase name="simulationControls">
  <SimulaBase name="integrationParameters">
    <SimulaConstant name="defaultSpatialIntegrationLength" type="double" unit="cm">
      0.
    </SimulaConstant>
  </SimulaBase>
  <SimulaBase name="outputParameters">
    <SimulaBase name="probeAllObjects">
      <SimulaConstant name="run" type="bool"> 0 </SimulaConstant>
      <SimulaConstant name="startTime" type="double"> 0 </SimulaConstant>
      <SimulaConstant name="endTime" type="double"> 28 </SimulaConstant>
      <SimulaConstant name="timeInterval" type="double"> 28 </SimulaConstant>
  </SimulaBase>
</SimulaBase>
```

Environmental & Root parameters

These are two sections in which different parameters are described using constants, tables and stochastic objects. Names that contain the word "Parameter" are ignored by most (all?) output modules. When someone develops other modules for SimRoot other parameter sections can be added. Note that the parameters are generally quired with relative – not absolute time. For example the growth rate of a root point should be quired with the age of the growth point – not the current real time. The constants in this section can be replaced by a table or a stochastic object without reprogramming the model. Thus there is great flexibility in the parameter section to define the root system you want. For example for primary roots you may want the growth rate to be constant while for lateral roots you may want to use a table which reduces the growth rate to zero after some time.

Many modules use sensible defaults when a parameter is not specified. Thus parameters are optional. If not the model will give an error when a required parameter is missing. The parameters in this section totally depend on the modules you are using. Your best off adapting an existing file – and check the documentation of the modules that you are using.

Templates

As mentioned before – SimulaBase objects can have a objectGenerator associated with them. These objectGenerators are modules that are invoked when ever the database receive a request later in time than the previous request. The object generators generate new objects. For example objects that simulate a new plant, a new root branch or new root segments. The objectGenerators that SimRoot currently uses use a template for generating objects. These templates are defined in the input file.

Templates make the SimRoot input file more complicated than the input files of most other models – which tent to just define parameters. You are not required to change the existing templates. You can easily skip the rest of this section and just change parameters in the parameter section. However – templates allow you to run SimRoot in more or less complex modes. For example – templates allow you to include or exclude the carbon model – include or exclude the phosphorus model etc. Templates are like lego blocks: they allow you to define your simulation problem. So for one simulation study SimRoot can run in simple and (fast) mode while for the other simulation study other components are required.

To make it easy to include or exclude for example the phosphorus model – all the tags that are needed for the phosphorus model are included in one file using the previously described file inclusion tags. So one SimulaInclude tag can enable or disable the phosphorus model.

One could argue that with these templates and with the specification of different modules – the input file of SimRoot has become a new scripting language. I tried to avoid that (for example modules are specified but there is no way to specify arguments for those models). However – flexibility and scalability of SimRoot where important design goals. I would like users to run the model as is appropriate for their simulation problem.

Strictly speaking not only SimulaBase – but all SimulaX objects can have object generators associated with them too – but most likely you'll use them with a SimulaBase tag. SimulaBase will than function as a 'named container'.

A plant and a soil

SimRoot can simulate any number of plants (as the cpu power and especially the memory of your computer allows). You just have to include a plant section for each plant that you want to specify. This section needs a planting date – planting position and genotype. There should be a parameter section for the specified genotype in the root parameter section. You are free to give you plant any name.

Optionally you can include a soil section. This section is still very much under development, but will include the possibility to run Hopmans or Simuneks 3D water and solute transport model.

Structure of SimRoot

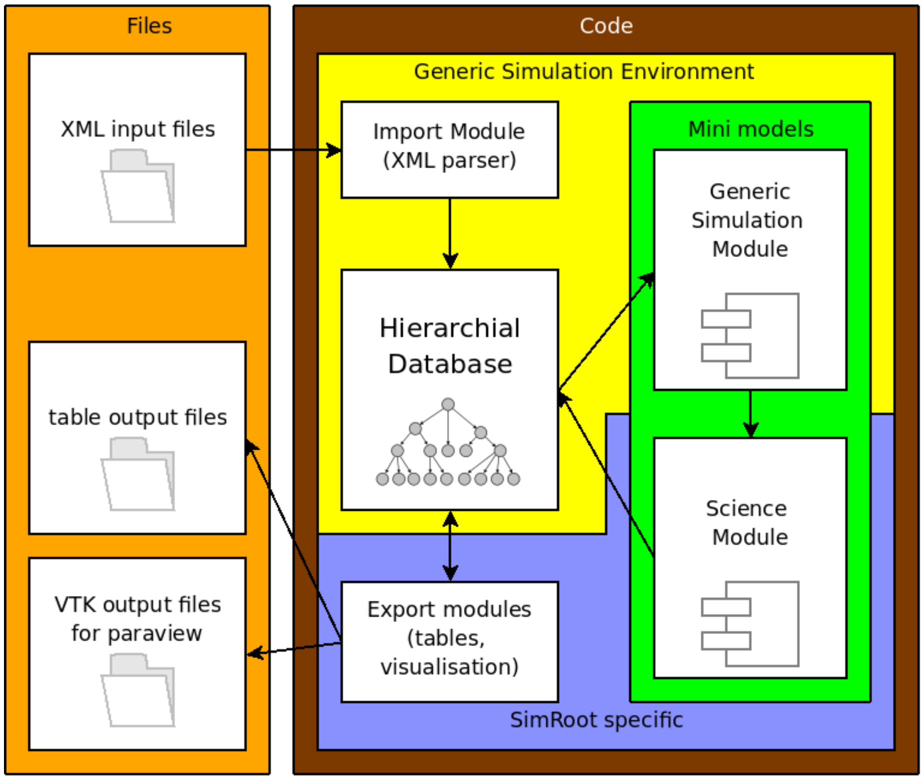

Files
XML input files
table output files
VTK output files for paraview
Code
Generic Simulation Environment
Import Module (XML parser)
Hierarchial Database
Mini models
Generic Simulation Module
Science Module
Export modules (tables, visualisation)
SimRoot specific

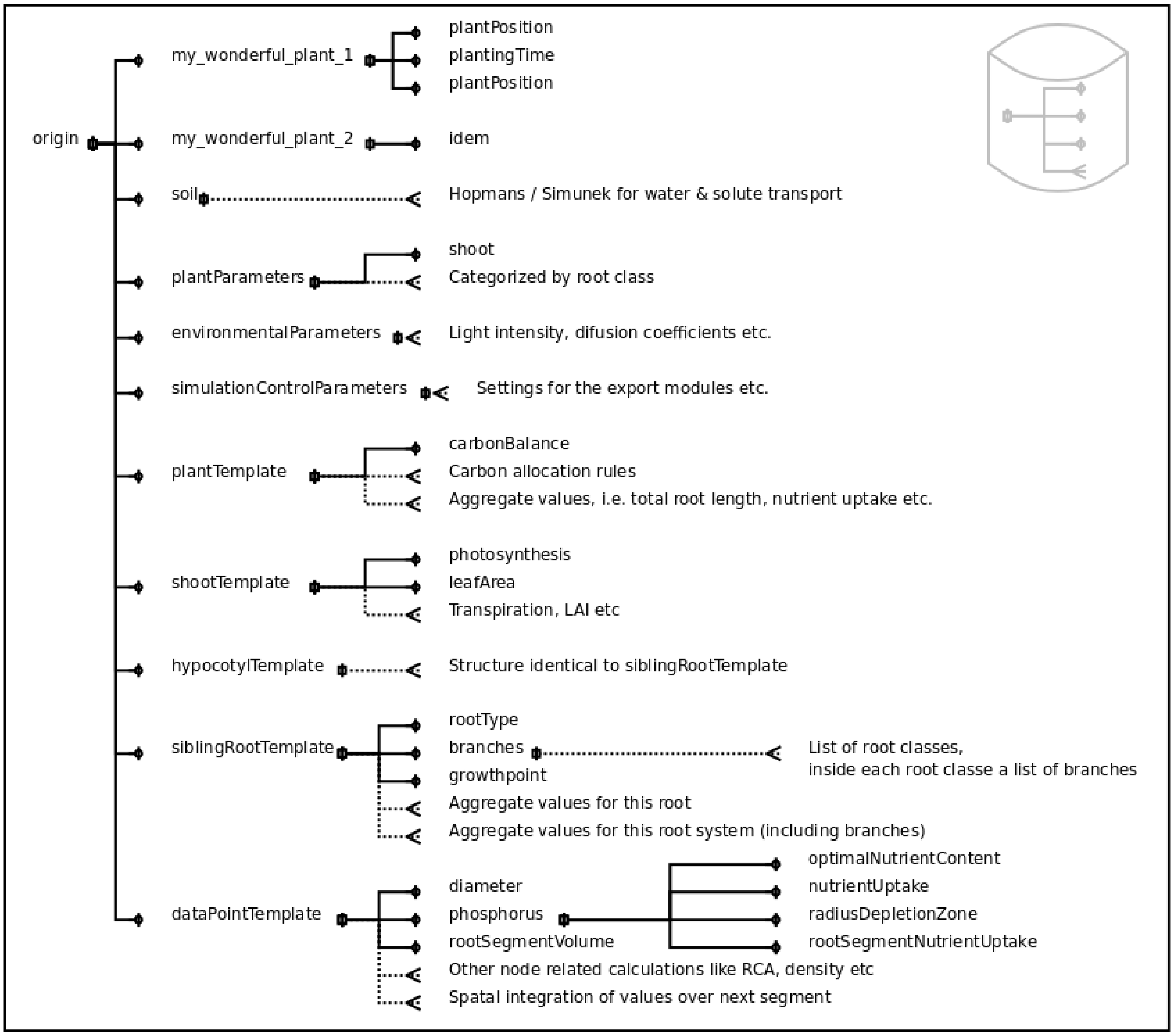
Figure 4: SimRoot's structure

SimRoot has four major components:

1) A set of input files, using XML formatting and an input module which will parse the XML.
2) A generic simulation environment consisting of database and generic simulation modules.
3) A function library which contains the 'science' part
4) An export library which writes the output files.

I already described the input and output files. But before I explain the code I would like to explain the hierarchial structure of the database and input files. The database and input files have the same hierarchical structure. This hierarchical structure gives context for different modules and mimics in some ways the branching structure of a root system.

Figure 5: The hierarchy in the input files and the database



origin
my_wonderful_plant_1
plantPosition
plantingTime
plantPosition
my_wonderful_plant_2
idem
soil
Hopmans / Simunek for water & solute transport
plantParameters
shoot
Categorized by root class
environmentalParameters
Light intensity, difusion coefficients etc.
simulationControlParameters
Settings for the export modules etc.
plantTemplate
carbonBalance
Carbon allocation rules
Aggregate values, i.e. total root length, nutrient uptake etc.
shootTemplate
photosynthesis
leafArea
Transpiration, LAI etc
hypocotylTemplate
Structure identical to siblingRootTemplate
siblingRootTemplate
rootType
branches
growthpoint
Aggregate values for this root
Aggregate values for this root system (including branches)
List of root classes,
inside each root classe a list of branches
dataPointTemplate
diameter
phosphorus
rootSegmentVolume
Other node related calculations like RCA, density etc
Spatal integration of values over next segment
optimalNutrientContent
nutrientUptake
radiusDepletionZone
rootSegmentNutrientUptake

The diagram shows this hierarchy. Note that if you, for example, do not include the carbon module, elements that belong to that module will not be part of the hierarchy. Some modules will have optional dependencies and behave differently depending on the presence of other modules.

It's important when you start coding a new module that you understand it's location in the hierarchy and the dependences on other modules.

Understanding the code

By now you should be able to run SimRoot and make changes to the input file. You should have discovered that the input file allows for great flexibility in defining your simulation problem. You should be aware that parameters can be change from constants to interpolation tables or random numbers without changing the code. Despite this flexibility you may not like the behavior of one or more modules or you want to add functionality. In this case – you will have to program a new module. Modules are registered C++ classes. You can link inside a module to other languages in case you want to link to for example another model written in Fortran. For this section I will assume you have basic understanding of C++. You should understand (object oriented) concepts and terms like class, inheritance, and pointers.

SimRoot links to the standard template library (stl) and makes heavily use of the stl containers. So you will have to know how to use these. Furthermore, – SimRoot depends on the boost library – but unless you would like to use the boost library in your own module – you don't need to know anything of this library. The SimRoot API does not expose the dependency on boost. (As of writing only the random number generators of boost are used – Annette Dathe wants to use the multi dimensional containers for the water and solute transport module).

Checking out and compiling the code

The SimRoot code is kept on a server with subversion control. Subversion allows us to keep a log book and to work with several people simultaneously on the same code. You can check out the subversion website.

SimRoot is currently an eclipse project. To work on SimRoot with eclipse – make sure you have the right version of eclipse (as of writing 3.4) – and the cdt, subeclipse, xmlauthor and photran plugins installed. SimRoot is developed on linux (Ubuntu hardy currently). Necessary software includes gcc, g95, and gdb and the make utilities. Necessary libraries are the standard c++ libraries and the random number generator from boost. For compiling and linking the Fortran code you will also need the gfortran runtime library (if g95 is you compiler).

Strictly speaking you do not need eclipse installed to work with simroot – but the make files are not included on the server – since eclipse will generate them automatically – so you may want to copy the make files from a system that does have eclipse installed. Usually I do this when I compile SimRoot on the PennState clusters.

Adding code instead of changing code

SimRoot consists of a general simulation environment and a library of scientific and export modules (See also the figure on SimRoot's structure). Most likely, you will only have to add a module to the scientific library or to the export modules. You can add these modules to SimRoot without knowing the details of the rest of the code.

I want to emphasize the word *add.* A lot of scientists tend to *change* code instead of *add.* The problem with that is that it is really hard to keep track of the 'who why when' questions of those changes (despite version control). Changing code will generally change the behavior of the model – and as a result break backwards compatibility and repeatability of previous experiments. The code that was used for many published papers is now lost because of untraceable changes. Many scientists have resided to keeping backups of old versions – but this leads to a chaotic number of code versions of which nobody knows what is used for what.

So my goal is to not change the behavior of SimRoot at all. Or in other words when in 5 years time I want to repeat a simulation from today – I just use the new executable with old input files (which indeed need to be backed up – but which are much smaller and can easily be kept with the backup of the project they were used for – not mentioning the comment in the input file which are more project specific than the comments in the code). To achieve this we need to stick to the 'add' paradigm. So if you want to change code in a module – you create a copy of the modules code – register it under a different name and make your changes in the copy – or you make your change a non-default option which depends ons the presence of other parameters. Thus we'll create many modules that do similar things (for example simulate leaf area) – but guarantee backwards compatibility. Even when you just want to correct a bug? I would say yes – if that module is used for a publication or important project. You could register the version with the bug correction under the same name with a version extension.

Some changes to the engine may break the backward compatibility – but in most cases will not. If they break – it generally a simple change in terms – which is easily corrected in the old input files. Changes to the engine should not change the core behavior of SimRoot though.

SimRoots workflow

SimRoot is made up of many independent modules. These modules request data from each other. How that data is simulated is hidden. Most models are driven by a large time loop, SimRoot is not. In SimRoot requesting data drives the model. Initial requests come from the export modules. Further requests often are generated by modules that depend on other modules. It's possible that, when for example the total volume of the rootsystem is not requested in the output, and not requested by the dependent modules that it is never calculated. In that sense, SimRoot is minimalistic.

The different modules share an identical structure. Essential, there are 4 different module types that inherit from 4 different base classes (SimulaBase, DerivativeBase, IntegrationBase, ObjectGeneratorBase). The following figure presents a work flow diagram which shows the basic operation and dependencies between the different modules in SimRoot.

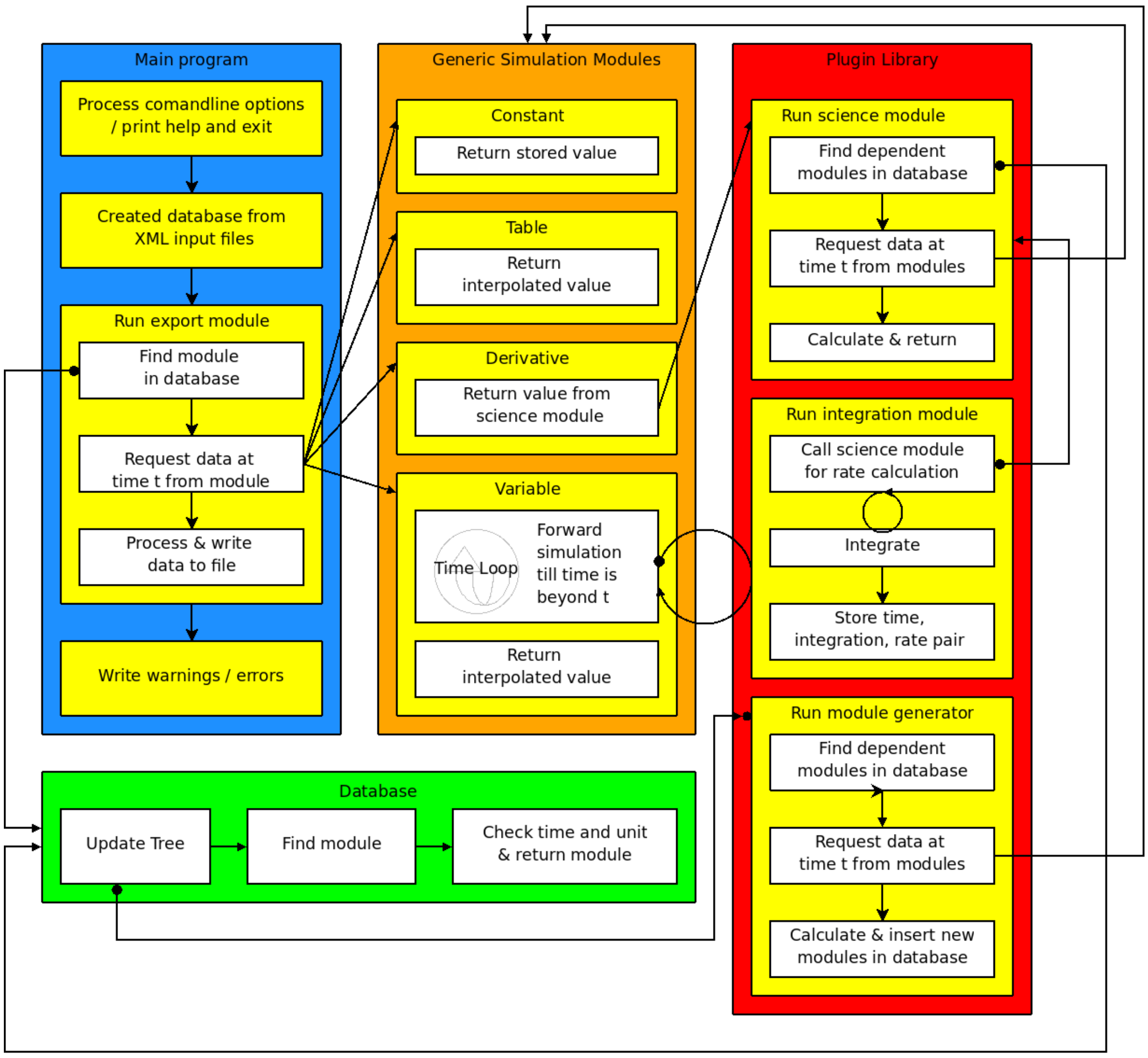

Main program
Process comandline options / print help and exit
Created database from XML input files
Run export module
Find module in database
Request data at time t from module
Process & write data to file
Write warnings / errors
Generic Simulation Modules
Constant
Return stored value
Table
Return interpolated value
Derivative
Return value from science module
Variable
Time Loop
Forward simulation till time is beyond t
Return interpolated value
Plugin Library
Run science module
Find dependent modules in database
Request data at time t from modules
Calculate & return
Run integration module
Call science module for rate calculation
Integrate
Store time, integration, rate pair
Run module generator
Find dependent modules in database
Request data at time t from modules
Calculate & insert new modules in database
Database
Update Tree
Find module
Check time and unit & return module

Figure 6: Workflow diagram for SimRoot

A few comments on this figure are appropriate.

1) It is possible to create circular dependencies. SimRoot will give an error message when you create a circular dependency.
2) The time loop is inside the modules. Unlike many models, SimRoot keeps dependencies at the larger structure and runs timeloops at a smaller scale. This means that different processes can run at different timesteps and, if independent, asynchronous. Interpolation solves the differences in time between modules.
3) The integration module supports iterative an predictor-corrected methods. The model does this by using a structure for data which combines each number with a flag. The flag indicates if a specific number is (base on) an estimate/predictor or not. The flag is preserved across calculations and modules.
4) The generic modules inherit from SimulaBase. The classes here correspond to the tags in the

input file. The database has the same structure as the input files.

5) The science modules inherit from DerivativeBase, the Integration modules inherit from IntegrationBase and the export modules inherit from ExportBase.

I will now discuss the different base classes and give you some guidance for creating your own module.

Modules in the plugin library

Plugins are self registering classes that inherit from a base class which is part of the engine (Strictly speaking are dynamically linked but for now we will ignore that detail, SimRoot uses static linking for its modules). There are currently three base classes - DerivativeBase, IntegrationBase and ObjectGenerator. The easiest way to add modules to the library is by copying the code of an existing class and than adjusting one. So here follow some examples.

DerivativeBase

Below is a header file for a class leafArea. This class simulates the leaf area of a plant. As you can see it inherits from DerivativeBase which is declared in SimulaDynamic.hpp. Besides the constructor, the class overrides two virtual methods from DerivativeBase: the initialize() method and calculate() method.

The class declares some protected variables and pointers. Generally these are data that will not change during dynamic simulation and can therefor be set in the initialization part. This particular classes saves two pointers to other objects that simulate required data. It also saves the time the plant was planted at.

```
#ifndef RATEFUNCTIONLIBRARY_HPP_
#define RATEFUNCTIONLIBRARY_HPP_

#include "../../SimulationClasses/SimulaDynamic.hpp"
class LeafArea:public DerivativeBase {
protected:
	virtual void initialize();
	virtual void calculate(const Time &t, State&);
	SimulaBase *SLA, *c2l;
	Time plantingTime;
public:
	LeafArea(SimulaDynamic* const pSV);
};
#endif
```

The next listing shows the c++ file. It contains the constructor, initialization and calculate methods for class leafArea. It also contains two functions and a class for auto registration of the leafArea class (plugin).

The base class provides a SimulaDerivative pointer named pSD which is your starting point for database operations, unit checking and more less used operations. The module starts off by checking it's units. This is a consistency check. If you specified this module in the input files with another unit the model will quit with an error message. Next we find other modules in the database. Finding modules is a matter of simply walking the pointer tree. GetParent will move you up the tree. Use an optional integer to move up several places. Use getChilderen to get a list of childeren of that module and use get to find a specific child. Childeren are recognized by name and you can optionally include the time, which will trigger an update of the list of childeren, and or the unit, which will check the expected unit of that module.

We do all database queries in the initialization section. This makes the model a lot faster – and is always allowed unless the existence of the object that you are looking for depends on the time. Note that macros are provided for common queries such as finding the plant type and plant parameter section. You can use the same macros.

```
#include "LeafArea.hpp"
#include "../../Toolkit/Messages.hpp"
#include "../../SimulationClasses/Origin.hpp"
#include "../PlantType.hpp"
#include "../../DataDefinitions/Units.hpp"
#include "../CarbonModule/Constants.hpp"

LeafArea::LeafArea(SimulaDynamic* const pSD):DerivativeBase(pSD){};
void LeafArea::initialize(){
	//check if unit given in input file agrees with this function
	pSD->checkUnit("cm2");
	//planting time
	pSD->getParent(3)->getChilderen()->get("plantingTime")->get(plantingTime);
	//plant type
	std::string plantType;
	PLANTTYPE(plantType,pSD)
	//plant parameters
	SimulaBase *param(ORIGIN.getChilderen()->get("rootTypeParameters")->
			getChilderen()->get(plantType));
	SLA=par->getChilderen()->get("shoot")->getChilderen()->
			get("specificLeafArea","g/cm2");
	c2l=pSD->getParent()->getChilderen()->get("carbonAllocation2Leafs","g");
};
void LeafArea::calculate(const Time &t, State&r){
	//local time
	Time localTime(t - plantingTime);
	//get portion of shoot allocated carbon that is used for leafs.
	c2lSimulator->getRate(t,r);
	//get specific leaf area SLA
	State sla;
	SLASimulator->get(localTime,sla);
	r/=sla;
	//multiply and return the result - assume that carbon to drymatter ratio is 0.54
	r/=CinDRYWEIGHT;
	if(r<0) msg::error("LeafArea:Negative leaf growth");
};

//Auto registration of the leaf area class
DerivativeBase * newInstantiationLeafArea(SimulaDynamic* const pSD){
   return new LeafArea(pSD);
};
class AutoRegisterRateClassInstantiationFunctions {
public:
   AutoRegisterRateClassInstantiationFunctions() {
	  derivativeBaseClassesMap["leafArea"] = newInstantiationLeafArea;
   };
};
//our one instance of the proxy
static AutoRegisterRateClassInstantiationFunctions p45564923142446542434;
```

The calculate section is straight forward on should be clean code with simple math formulating your scientific formula. In this case the leaf area is calculated from the carbon that is allocated to the leafs an the specific leaf area. A carbon to dryweight correction factor is used which is present in the model as a macro constant.

Finally the model check the result. You have the option to use warnings or errors. Be strict with errors. Don't allow the model to continue when results really don't make sense. If you use an warning when there should be an error – probably the error will be generated somewhere else – confusing you and other users of what went wrong. Warnings should only be issued when there is an appropriate fall back.

The autoregistration part you do not need to understand. Just replace LeafArea with the name of your class. Make sure however that you choose a clear and unique name in “derivativeBaseClassesMap["nameOfMyFunction"]”. So if you wanted to calculate the leafArea differently – you should register it under a different name. For example “leafAreaHopkinsVersion0.1”. See the discussion on changing versus adding code above.

IntegrationBase

You really should not try to write a new integration method unless you understand the data flow in SimRoot thoroughly. If you do, I am sure you could adapt a copy of one of the current integration methods.

ObjectGeneratorBase

The object generator is meant to generate new objects. You need to override the initialization and generate methods. Below listing give a shortened example, including the now familiar auto registration part. The base class provides a SimulaBase pointer named pSB which is your starting point for database operations.

It can be tricky to write the generate method. You do not only need to know where to generate what but also when. Each SimulaPoint Object has a starting and ending time. It is important that these are set correctly. Otherwise the model may give inconsistent results. For example – when you create root branches you should set the time of creation of these root branches correctly. Also do not assume that you only have to create one set of branches. Any length of time could be between the previous call and the current call to this function. So you may have to create a loop inside your root branching module that will create as many branches as fits the branching parameters.

It is quite possible that after a certain condition your objectGenerator does not need to generate any more objects. For example after maximum number of branches is reached – no more branches will be created till the end of the simulation. To stop the calls to the objectGenerator you can include the following code: `pSB->getChilderen()->stopUpdatefunction();`

shortened header file

```
#ifndef ROOTBRANCHING_HPP_
#define ROOTBRANCHING_HPP_

#include "../../SimulationClasses/Database.hpp"
class RootBranches:public ObjectGenerator{
private:
	//you can optionally declare some private variables here
public:
	virtual void initialize(const Time &t);
	virtual void generate(const Time &t);
	RootBranches(SimulaBase* const pSB);
};
#endif /*ROOTBRANCHING_HPP_*/
```

shortened C++ file

```
//Your code for initialize(t) and generate(t) here

//the maker function for instantiation of the class
ObjectGenerator * newInstantiationRootBranches(SimulaBase* const pSB){
   return new RootBranches(pSB);
};

class AutoRegisterRootBranchingInstantiationFunctions {
public:
   AutoRegisterRootBranchingInstantiationFunctions() {
      // register the maker with the factory
          objectGeneratorClassesMap["rootBranches"] = newInstantiationRootBranches;
   };
};

AutoRegisterRootBranchingInstantiationFunctions
oneInstanceForRegistrationOfObjectGeneratorInstantiationFunctions;
```

The export modules

The export modules are as of writing not self registering plug-ins. It's on my todo list to change that. However, they are classes that derive from ExportBase. The export classes are instantiated in a function runExportModules(int); So to add and export module you will have to add a call to it in this function.

Below is the declaration of the base class. You will have to override the run() method.

```
class ExportBase{
public:
       ExportBase(std::string module);
       virtual ~ExportBase();
       virtual void go();
protected:
       virtual void run()=0;
       std::string moduleName;
       Time startTime, endTime, intervalTime;
       SimulaBase* controls;
       bool runModule;
};
```

The run method most likely should include a time loop which starts at startTime, goes with intervals of intervalTime and stops when endTime is reached. An example is given below. The example also prints a progress indicator on the screen. For your convenience a macro provided which implements the example time loop.

Your export module may need user specified settings. Each export module has therefor a section in the input files – as described before. The SimulaBase pointer 'controls' points to the location in the database that corresponds to this section.

```
void ProbeAllObjects::run(){
       //run probing loop
       unsigned int progress(0);
       for (Time t=startTime ; true ; t+=minimum(intervalTime,endTime-t)){
              //probe model
              probeObject(t,&ORIGIN);
              //run progress indicator
              progress=int(100*(t-startTime)/(endTime-startTime));
              std::cout<<ANSI_save<<progress<<"%"<<ANSI_restore<<std::flush;
              if(t==endTime) break;
       };
};
```

Simulation environment

The simulation environment is made of a couple of simulation classes. The simulation classes all inherit from SimulaBase – and more important for you – all classes that do some dynamic simulation – classes which you are planning to write modules – inherit from SimulaDynamic which in its place inherits from SimulaBase. You are already familiar with what these classes do – since they correspond 1 to 1 with the simulation tags in the input file. So there is a class for constants, interpolation tables, etc.

The simulation classes have different ways of simulating data – but have mostly the same set of methods (virtual methods that are present in SimulaBase – or SimulaDynamic). Any class specific method is most likely not needed for writing your module. It is possible that you want to add a method to a class – than you should probably add this method to SimulaBase or SimulaDynamic too, if you want to avoid type casting. When methods are not appropriate (for example get(time,point) for a class that does not simulate point data) an error is thrown.

The most common used method is get(t,data) where t is the time you want data at, and data is the variable that on return has the value you requested. Data can be of several different types, currently int, Time, State, and Coordinate are supported. So the following code would print 0,0,0:

```
extern SimulaPoint ORIGIN;
Coordinate position;
ORIGIN.get(t,position);
std::cout<<position;
```

All instantiations from the SimulaX classes get automatically registered in the database. This database is no more than a tree of pointers. At the top of the tree you will find the origin. Each simulation object has methods for moving up to a parent in the tree or for finding the list of children. Thus you can traverse the database starting relative from where your current position is. Alternatively you can traverse the database starting at the origin and than moving down. A few examples may help:

```
SimulaBase* goal=
   origin.getChilderen()->get(t,”origin's Child”)->getChilderen(t,”origin's GranChild”);
SimulaBase* relative=current->getParent()->getChilderen()->get(t,”current's Sibling”);
```

(ps. Yes there is a spelling error in children – which I hope to correct soon)

The classes that inherit from SimulaDynamic make use of the function library. The function library is a collection of classes that contain the scientific calculations needed for the simulation. Most of the function library is quite straight forward – for example a function could calculate the length of a root segment – or sum up the length of all root segments. Some functions will calculate the derivative which is than integrated by the simulation object. For example – the growth rate is integrated so you get root length.

SimulaPoint objects simulate the movement of a point in time. For example the tip of a root. To simulate this the direction and speed of the point are calculated each time step and integration is used to determine the new position.

Basic data types

You need to know a little about the basic data types that SimRoot is using. These are declared in the header files in the DataDefinitions folder. These data types are currently: State, StateRate, Coordinate, MovingCoordinate, Time and Unit.

State is a structure that you should use instead of where you would use double or float. State carries a double with a boolean. This boolean – named estimate – will be preserved in any arithmetics – and will flag the engine if the returning result is an initial estimate or the final result. These initial estimates may be a result when iterative integration methods are used. Normally you do not need to worry about the state of the estimate flag – just make sure it is preserved by not using doubles and floats in your class. However, when you want to use specific function from for example the math library, you should tag that the flag is preserved and if not, you should preseve it yourself. The estimate flag of the result should be non-zero if any of the flags of the input variables were non-zero.

StateRate is made off two states. It is used to store a value and it's derivative. Both have estimate flags.

Coordinate stores xyz coordinates Negative Y is down in SimRoot – similar to most 3D rendering environments and similar to charts. However – be aware that other coordinates systems do exist in other models. Coordinate also carries an estimate flag. The unit of the coordinate system is in cm.

MovingCoordinate, like StateRate, stores two coordinates – the position of a point and the derivative (speed and direction) of a point.

Units stores the name, type and conversion factor of standard units. You can do string comparison and arithmetics on units. For example:

```
Unit a(“cm”),b(“day”);
Unit c(a/b);
std::cout<<c.name;  //prints cm/day
if(c==”cm/day”) std::cout<<”c has the correct unit”;
double n=getUnitConversionFactor(a, “m”);
```

Time is currently just a double from beginning of simulation. Default time in the model is in days.

Description of the modules in SimRoot

Sorry, there is no module reference right now. You'll just have to understand it from the code. Really, most modules are pretty simple and straight forward.

Disclaimer

This manually probably contains errors and simroot is in constant development. So it may be outdated. But I hope that much of the 'big picture' remains true for some years to come.

Jouke Postma