Simulation de la croissance de plantes en communauté dans un écosystème prairial

Benjamin Gouriou

Institut **S**upérieur d'**I**nformatique
de **M**odélisation et de leurs
Applications

Campus des Cézeaux
BP 125
63173 Aubière Cedex

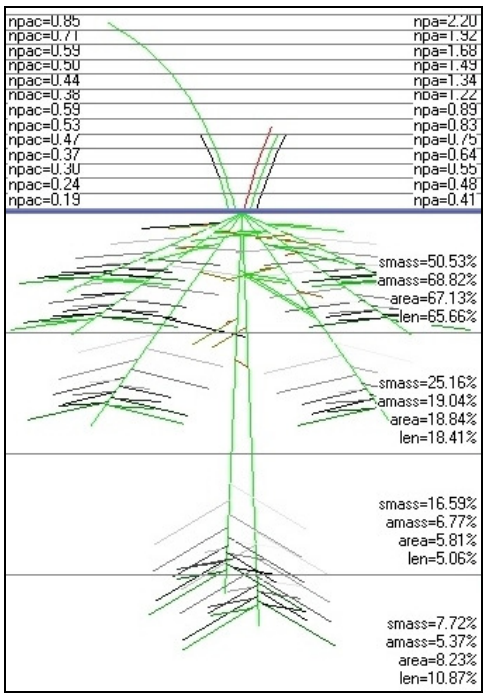
Institut **N**ational de **R**echerche
Agronomique de Clermont-Ferrand

Site de Crouël
234 Avenue du Brézet
63000 Clermont-Ferrand

Rapport de Stage de deuxième année

Filière 3 : Informatique des Systèmes d'Information et de Production et Aide à la Décision

Simulation de la croissance de plantes en communauté dans un écosystème prairial



Présenté par : _**GOURIOU Benjamin**
Maître de stage : _**MARTIN Raphaël**
Tuteur ISIMA : _**LACOMME Philippe**

Avril 2009 – Septembre 2009

Institut **S**upérieur d'**I**nformatique
de **M**odélisation et de leurs
Applications

Campus des Cézeaux
BP 125
63173 Aubière Cedex

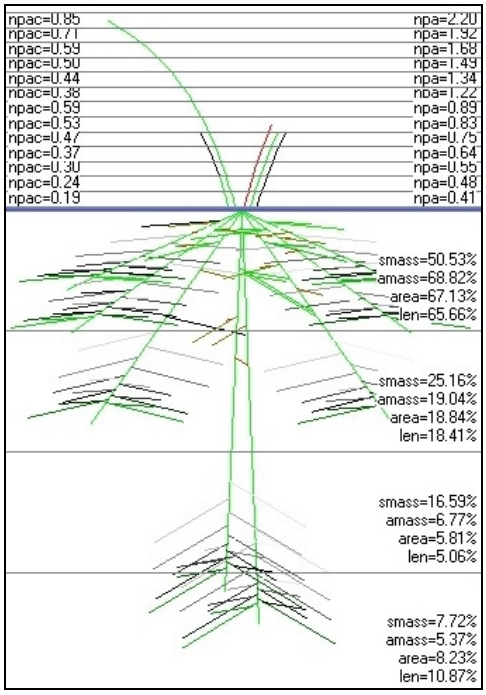
Institut **N**ational de **R**echerche
Agronomique de Clermont-Ferrand

Site de Crouël
234 Avenue du Brézet
63000 Clermont-Ferrand

Rapport de Stage de deuxième année

Filière 3 : Informatique des Systèmes d'Information et de Production et Aide à la Décision

Simulation de la croissance de plantes en communauté dans un écosystème prairial



Présenté par : _**GOURIOU Benjamin**
Maître de stage : _**MARTIN Raphaël**
Tuteur ISIMA : _**LACOMME Philippe**

Avril 2009 – Septembre 2009

Remerciements

Je tiens tout d'abord à remercier l'ensemble de l'équipe de l'unité UREP (Unité de Recherche sur l'Ecosystème Prairial) pour son accueil chaleureux. Je souhaite aussi remercier Jean-François SOUSSANA sans qui ce stage n'aurait pas eu lieu.

Je remercie plus particulièrement Raphaël MARTIN, mon maître de stage, pour sa disponibilité, sa patience lors des nombreuses relectures de ce rapport ainsi que pour toute l'aide et les conseils qu'il a su m'apporter tout au long de ces six mois.

Mes remerciements vont ensuite à Vincent MAIRE, qui a réalisé sa thèse sur GEMINI. C'est en effet grâce à ses explications et à la pédagogie dont il a fait preuve à mon égard que j'ai pu aborder sereinement la phase de modélisation de l'eau. Enfin, je tiens également à remercier David HILL et Luc TOURAILLE pour leur aide quant à l'architecture et l'implémentation du programme.

Résumé

Depuis plusieurs années, l'Unité de Recherche sur l'Ecosystème Prairial de l'INRA de Clermont-Ferrand développe un **modèle** d'écosystème prairial nommé GEMINI. Ce modèle a été développé au sein d'une plate-forme de simulation **générique** écrite en **C++** et baptisée UNIF. Ce modèle est utilisé par deux équipes de recherche, l'UREP et l'équipe de la Max Planck Institute à Jena en Allemagne.

Chacune de ces équipes poursuit des objectifs à court terme différents ce qui a aboutit à deux **versions** évoluant en parallèle. Afin de faciliter les développements du modèle et les échanges inter-équipes, les scientifiques ont besoin d'un unique modèle regroupant les fonctionnalités disponibles au sein de ces deux versions, tout en rendant possible le lancement d'une simulation sous une version spécifique. L'architecture du modèle a donc été modifiée de manière à pouvoir **mutualiser** au maximum les méthodes et attributs communs aux deux versions tout en permettant à l'utilisateur de choisir la version avec laquelle il lance une simulation.

D'un point de vue fonctionnel, GEMINI devait évoluer afin de pouvoir modéliser au mieux le fonctionnement de la prairie. Nous avons donc intégré à la modélisation les **flux d'eau** dans le continuum sol-plante-atmosphère. L'intégration de ce mécanisme nécessite un travail important de réflexion si l'on veut fondre proprement cette modification à l'architecture existante de GEMINI. Aussi en concertation avec plusieurs biologistes, mon rôle a été de préparer la phase d'implémentation en testant la robustesse des équations proposées puis en réfléchissant à différentes méthodes d'optimisation.

Mots-clés : modèle, générique, C++, versions, mutualiser, flux d'eau

Abstract

For several years, the Research Unit on Grassland Ecosystem at INRA, Clermont-Ferrand has been developing a grassland ecosystem **model** called GEMINI. This model runs on a **generic** simulation framework written in **C++**, called UNIF. This model is used by two research teams, UREP and a team of the Max Planck Institute of Jena in Germany.

Each of these teams has their own short term research objectives, and as a result two **versions** of GEMINI have been developing in parallel. In order to facilitate new developments of the model and exchanges between the two teams, scientists need a unique model grouping the two versions' functionalities. My work consisted of modifying the architecture of the model so as to **share attributes** and allows users to choose which version of GEMINI they want to work with.

From a biological point of view, GEMINI needed to evolve to better simulate the grassland functioning. We integrated a new option in the model: **water flows** in the soil-plant-atmosphere continuum. The addition of this module needs considerable research and organization to integrate it in GEMINI's architecture. Then, in agreement with the biologist of the team, my role was to prepare the implementation phase by testing the proposed equations and by thinking to different optimisation methods.

Keywords: model, generic, C++, versions, share attributes, water flows

Lexique

Dicotylédone : Classe de plantes qui présentent une plantule à deux cotylédons (feuilles embryonnaires) lors du développement de la graine.

Graminées (grass) : Famille de plantes monocotylédones très présentes dans les milieux prairiaux, aux fleurs peu apparentes et groupées en épis.

IDE : **I**ntegrated **D**evelopment **E**nvironment (ou environnement de développement intégré) est un programme regroupant un ensemble d'outils pour le développement de logiciels. Il contient généralement un éditeur de texte, un compilateur et un débogueur et est le plus souvent dédié à un seul langage de programmation.

Légumineuses (legume) : Famille de plantes dicotylédones dont la particularité est de capter l'azote atmosphérique grâce à une symbiose avec des bactéries présentes dans les nodosités fixées à ses racines. Cette propriété en fait une famille très importante au niveau agronomique puisqu'elles permettent d'enrichir les plantes et le sol en azote.

Matière humifiée : Matière organique stable du sol provenant de la décomposition de la litière végétale et de la biomasse morte microbienne et animale essentiellement par l'action combinée des animaux, des bactéries et des champignons du sol.

Rapport C/N de la matière organique : le rapport carbone sur azote est un indicateur qui permet de juger du degré d'évolution de la matière organique, c'est-à-dire de son aptitude à se décomposer plus ou moins rapidement dans le sol.

RAD : **R**apid **A**pplication **D**evelopment (ou environnement de développement rapide) est un outil qui permet de développer un logiciel très rapidement et se caractérise généralement par un système de génération automatique de code.

Sénescence : Processus physiologique qui entraîne une lente dégradation des fonctions de l'organisme, synonyme de vieillissement.

STL : Standard Template Library est la bibliothèque standard du langage C++ et contient une multitude de classes et méthodes utiles aux développeurs. Cette bibliothèque est normalisée par ISO depuis le début des années 1990.

Stomate : Cavité de petite taille présent dans l'épiderme des feuilles d'une plante. Il permet les échanges gazeux entre la plante et l'air ambiant (dioxygène, dioxyde de carbone, vapeur d'eau...)

Stress hydrique : Etat d'une plante quand la ressource en eau est insuffisante pour satisfaire la demande.

Unicode : Norme informatique, qui vise à donner à n'importe quel caractère un nom et un identifiant numérique, et ce de manière unifiée, quelle que soit la plate-forme informatique ou le logiciel.

VCL : **V**isual **C**omponent **L**ibrary (ou bibliothèque de composants visuels) est un ensemble de bibliothèque de composants graphiques, écrites par Borland et permettant de créer des interfaces graphiques sous Windows.

Table des matières

Remerciements
Résumé
Table des matières
Table des figures

Introduction 1

Présentation de l'INRA 3
1. Un organisme de recherche 3
2. Le centre de Clermont-Ferrand – Theix – Lyon 3
3. L'UREP 4

Introduction à l'étude 5
1. Analyse de l'existant 5
   1.1. UNIF, une plate-forme de simulation 5
   1.2. Le modèle GEMINI 10
2. Outils informatiques 11

Réalisation d'une version hybride des modèles existants 13
1. Problématique 13
2. Analyse de l'environnement de développement : Canopt 13
3. Etude des différentes possibilités d'implémentation 15
   3.1. L'utilisation d'un héritage conditionnel 16
   3.2. Utilisation de *template* pour les spécialisations 19
   3.3. Utilisation de l'agrégation à la place de l'héritage 21

Développements du modèle GEMINI 25
1. Développements annexes 25
   1.1. Génération et chargement de fichiers de paramétrage 25
   1.2. Evolution vers un outil plus récent 27
2. Intégration de l'eau 29
   2.1. Mécanismes de l'évapotranspiration 29
   2.2. Application des modèles et méthode de convergence 30

Perspectives et discussion 32

Conclusion 35

Références bibliographiques 36

Table des figures

Figure 1 : Plan d'accès à l'INRA, site de Crouël 4
Figure 2 : Diagramme UML simplifié du modèle SOILOPT 6
Figure 3 : Diagramme UML simplifié de la plate-forme UNIF 7
Figure 4 : Code de la méthode run() de la classe Model 8
Figure 5 : Relation entre Node et Model 9
Figure 6 : Interface d'UNIF 9
Figure 7 : Diagramme UML simplifié du modèle GEMINI 11
Figure 8 : Diagramme UML simplifié Canopt INRA 14
Figure 9 : Diagramme UML simplifié Canopt MERGE / JENA 15
Figure 10 : Paramètre de choix du modèle dans l'interface 16
Figure 11 : Solution 1 - Héritage conditionnel 17
Figure 12 : Extrait du code de l'héritage conditionnel, fonctionnement de Selector 18
Figure 13 : Diagramme UML simplifié de la solution par "template" 19
Figure 14 : Extrait du code de l'utilisation des « template » pour l'héritage 20
Figure 15 : Diagramme UML générique du patron AbstractFactory 21
Figure 16 : Diagramme UML simplifié de la version finale 22
Figure 17 : Extrait du code relatif au mécanisme "Factory" 23
Figure 18 : Extrait de code des membres static de ShootsMERGE 26
Figure 19 : Code de la fonction inspectClass() implémentée dans ShootsMERGE 26
Figure 20 : Interface de C++ Builder 2009 28
Figure 21 : Cycle de l'eau dans le sol et la plante 30
Figure 22 : Résultats de la méthode de convergence 31
Figure 23 : Comparaison des méthodes en temps CPU 32

Table des annexes

ANNEXE 1 : Les fichiers de paramétrage de GEMINI 37

ANNEXE 2 : Modèle de Sinclair..... 38

ANNEXE 3 : Modèle de Tuzet..... 40

ANNEXE 4 : Interface de Berkeley Madonna 41

Introduction

Dans le cadre de ma deuxième année d'étude à l'Institut Supérieur d'Informatique, de Modélisation et de leurs Applications, j'ai effectué un stage de cinq mois, d'Avril à Septembre 2007, au sein de l'Institut National de Recherche Agronomique de Clermont-Ferrand, sur le site de Crouël.

Depuis plusieurs années, l'INRA développe le modèle GEMINI (Grassland Ecosystem Model with INdividual centered Interactions). Ce modèle de simulation déterministe orienté-objet écrit en C++ permet de représenter l'évolution d'une population de plantes dans un écosystème de type prairie permanente gérée, et ce dans le but de mettre en évidence le rôle de la diversité et des interactions biotique (compétition des plantes pour les ressources du sol et pour la lumière) pour le fonctionnement des écosystèmes prairiaux.

Le modèle GEMINI est au cœur du projet ANR DISCOVER dont est chargée l'Unité de Recherche sur l'Ecosystème Prairial, en collaboration avec le Max Planck Institute de Jena en Allemagne. Un des principaux objectifs méthodologiques de ce projet est la confrontation du modèle de simulation à des expériences sur le terrain de manière à l'affiner au fur et à mesure.

Dans un premier temps je présenterai le modèle et ferai un état des lieux sur les travaux déjà effectués. Je décrirai ensuite ma démarche et les modifications que j'ai effectuées sur GEMINI avec tout d'abord la construction de la version « hybride » puis ensuite la modélisation des flux d'eau et les développements annexes au modèle. Enfin je discuterai des résultats de mes travaux ainsi que des perspectives ouvertes à l'issu de mon stage.

Présentation de l'INRA

1. Un organisme de recherche

Créé en 1946, l'Institut National de la Recherche Agronomique est un établissement public à caractère scientifique et technologique placé sous la double tutelle des ministères chargés de la Recherche et de l'Agriculture. Il a pour vocation la production, la diffusion et le partage de connaissances dans les domaines de l'agriculture, de l'alimentation et de l'environnement. C'est à l'heure actuelle le troisième organisme de recherche français après le CNRS et le CEA, avec un budget de 800,5 millions d'euros pour 2007.

L'INRA dispose de 20 centres régionaux répartis en 150 sites de recherche dans toute la France. Ceux-ci comportent 470 unités pour un effectif total de plus de 8500 personnes et accueillent chaque année 4000 stagiaires et doctorants.
Les 14 départements de recherche de l'INRA relèvent de 6 directions scientifiques nationales :

- Agriculture, activités et territoires.
- Animal et produits animaux.
- Environnement, écosystèmes cultivés et naturels.
- Nutrition humaine et sécurité alimentaire.
- Plante et produits du végétal.
- Société, économie et décision.

2. Le centre de Clermont-Ferrand – Theix – Lyon

Le centre de Clermont-Ferrand – Theix – Lyon compte 770 agents permanents dont 340 chercheurs et ingénieurs. Il regroupe l'ensemble des 30 unités de recherche et expérimentales implantées en Auvergne. Le Centre représente un fort potentiel pour l'Institut National de la Recherche Agronomique, puisqu'il rassemble 9% de l'effectif total de l'Institut. Ses thématiques de recherches sont multidisciplinaires et les 30 unités qui le composent relèvent de 13 départements de recherche sur les 14 que comprend l'Inra.

Les recherches couvrent différents domaines d'investigation allant de l'exploitation agricole, la prairie et l'animal, aux produits animaux (lait, fromages, viande) et à l'Homme, sans oublier le végétal. On peut les regrouper en 4 grands secteurs de recherche :

- L'élevage durable et l'environnement dans les zones herbagères de montagne.
- L'élaboration de la qualité des produits animaux : de l'herbe au fromage et à la viande.
- La nutrition humaine préventive et le vieillissement, l'impact de la « fonction signal » des aliments.
- L'écophysiologie et la génomique pour la qualité des produits des céréales et de l'arbre

3. L'UREP

Les travaux de l'Unité de **R**echerche sur l'**E**cosystème **P**rairial (UREP) concernent l'étude intégrée du fonctionnement écologique de la prairie pâturée, en relation avec la gestion agricole et avec le milieu physique. Ils sont appliqués à des objectifs de production et de protection de l'environnement : maîtrise de la dynamique de la végétation, entretien de l'espace et du paysage, gestion des milieux pour la conservation de la biodiversité, préservation des stocks de carbone dans les sols et limitation des pollutions azotées. Les études expérimentales analysent le fonctionnement des peuplements prairiaux et de leurs interfaces avec l'herbivore, le sol et l'atmosphère (gaz à effet de serre). La dynamique des écosystèmes prairiaux est modélisée numériquement dans un but de synthèse des connaissances et de prévision. Les programmes concernent des prairies en situation de sous-exploitation par le pâturage (comportant ou non des ligneux spontanés ou plantés), des associations entre espèces prairiales modèles (graminées et légumineuses) et des écosystèmes prairiaux soumis à des changements de la composition de l'atmosphère (CO_2) et du climat.

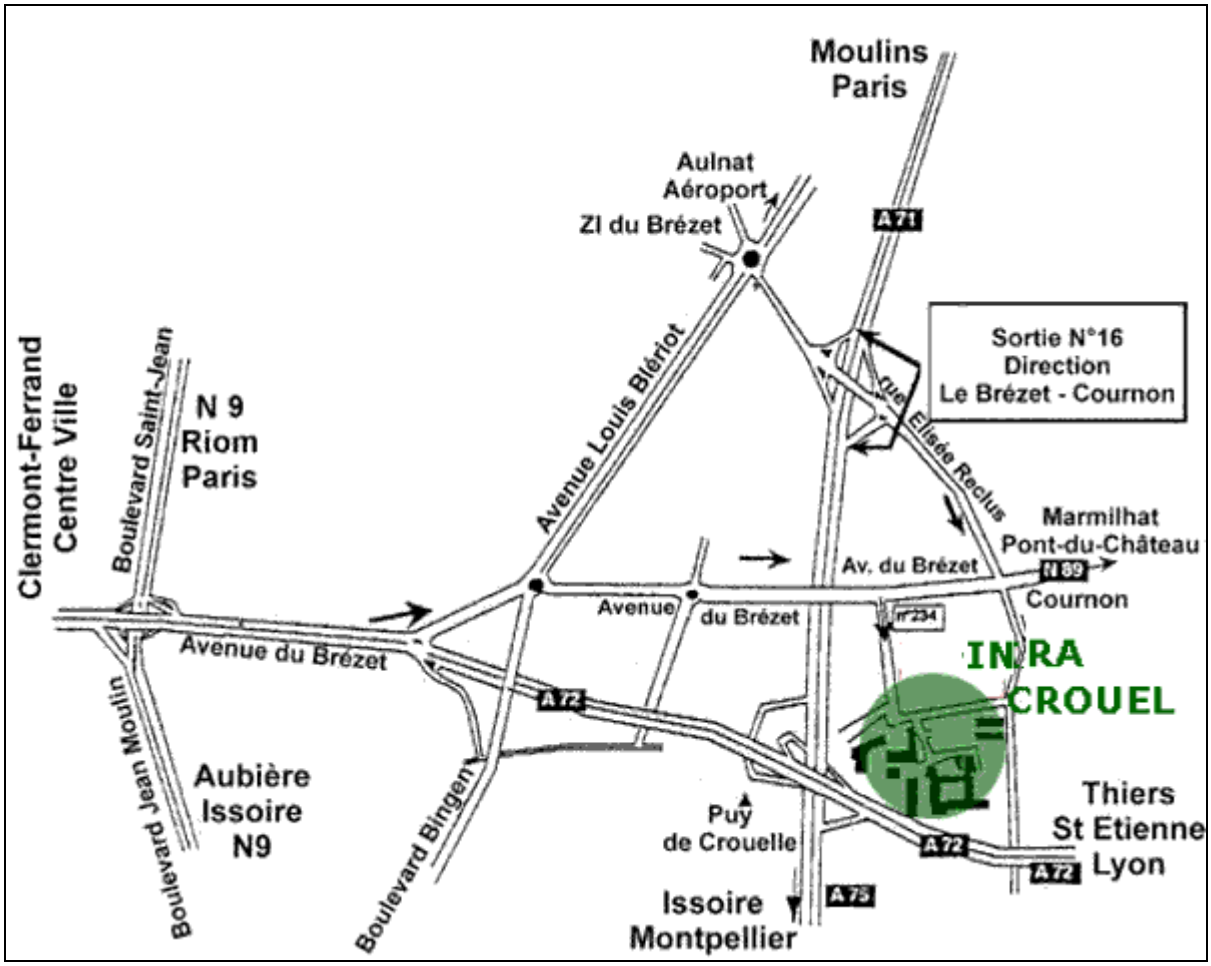


Figure 1 : Plan d'accès à l'INRA, site de Crouël

Introduction à l'étude

« Un modèle est une abstraction qui simplifie le système réel étudié pour se focaliser sur les aspects qui intéresse le modélisateur et qui définissent la problématique du modèle » [COQUILLARD et HILL 1997]. Pour les chercheurs, la création de modèles est donc un moyen de décrire la réalité et de mieux comprendre les mécanismes et propriétés qui la régissent. GEMINI (**G**rassland **E**cosystem **M**odel with **IN**dividual based **I**nteractions) est un modèle écophysiologique centré sur l'individu en lien avec le climat et le fonctionnement du sol. Il est composé de deux sous-modèles : CANOPT, qui représente une ou plusieurs populations de plantes, et SOILOPT, qui modélise le fonctionnement du sol. GEMINI a pour objectif de permettre de mieux comprendre le rôle et l'évolution de la biodiversité dans les écosystèmes prairiaux, en s'intéressant aux relations plante-plante à l'échelle de l'individu. Il s'agit d'un modèle de simulation déterministe, orienté-objet et développé en C++. Il est intégré sur une plate-forme de simulation nommée UNIF (**U**nified **N**umerical **I**ntegration **F**ramework) et permet de simuler le fonctionnement d'un écosystème prairial.

Ce modèle est utilisé par deux équipes de recherche : l'UREP au sein de laquelle je réalise mon stage et l'équipe de recherche du Max Planck Institute à Jena en Allemagne. Chacune de ces équipes poursuit des objectifs à court terme différents ce qui a aboutit à deux versions évoluant en parallèle. Si ces versions sont déjà portées sur un logiciel de gestion de version permettant d'encadrer leur divergence, il demeure toujours la difficulté de cumuler les acquis de chacune des versions en une seule. Ainsi, le premier objectif de mon stage est de proposer une solution permettant de comparer les versions et de passer facilement de l'une à l'autre.

Le modèle GEMINI est un modèle qui s'intéresse aux flux de carbone et d'azote entre les plantes et le sol. Afin de mieux appréhender la réalité des milieux écologiques, les chercheurs ont souhaité développer les flux d'eau dans le continuum sol-plante-atmosphère. Aussi, le second objectif de mon stage est de développer le modèle pour intégrer le développement biologique souhaité par les chercheurs de l'équipe.

Avant toute chose, je m'attacherai à présenter une étude de l'existant pour répondre au mieux à ces deux objectifs.

1. Analyse de l'existant

1.1. UNIF, une plate-forme de simulation

A l'origine, l'unité UREP avait conçu plusieurs modèles scientifiques, implémentés en C++ principalement par des étudiants de l'ISIMA dans le cadre de leur projet ou de leur stage.

- Le modèle SOILOPT, pour SOIL OPTimisation, fut développé à partir de 1998 par Pierre Loiseau (INRA) et Yannick Bergia (ISIMA). C'est un modèle de sol couplant les flux de Carbone et d'Azote. Il décrit la dynamique de 4 compartiments de matière organique de sol, chacun avec un rapport C/N fixe (DOM, EOM, IS, NS). Deux groupes de microbes, les bactéries (MBB) et les champignons (MBF), dégradent respectivement la litière fraîche (POM) et la matière organique humifiée (WF, WR) du sol. Les deux groupes microbiens diffèrent par leur taux potentiel de croissance, le type de substrat utilisé pour la croissance et leur besoin en azote. Cette description de la diversité microbienne permet de simuler la limitation énergétique de la décomposition des matières organiques récalcitrantes. La figure suivante présente le diagramme UML de SOILOPT.

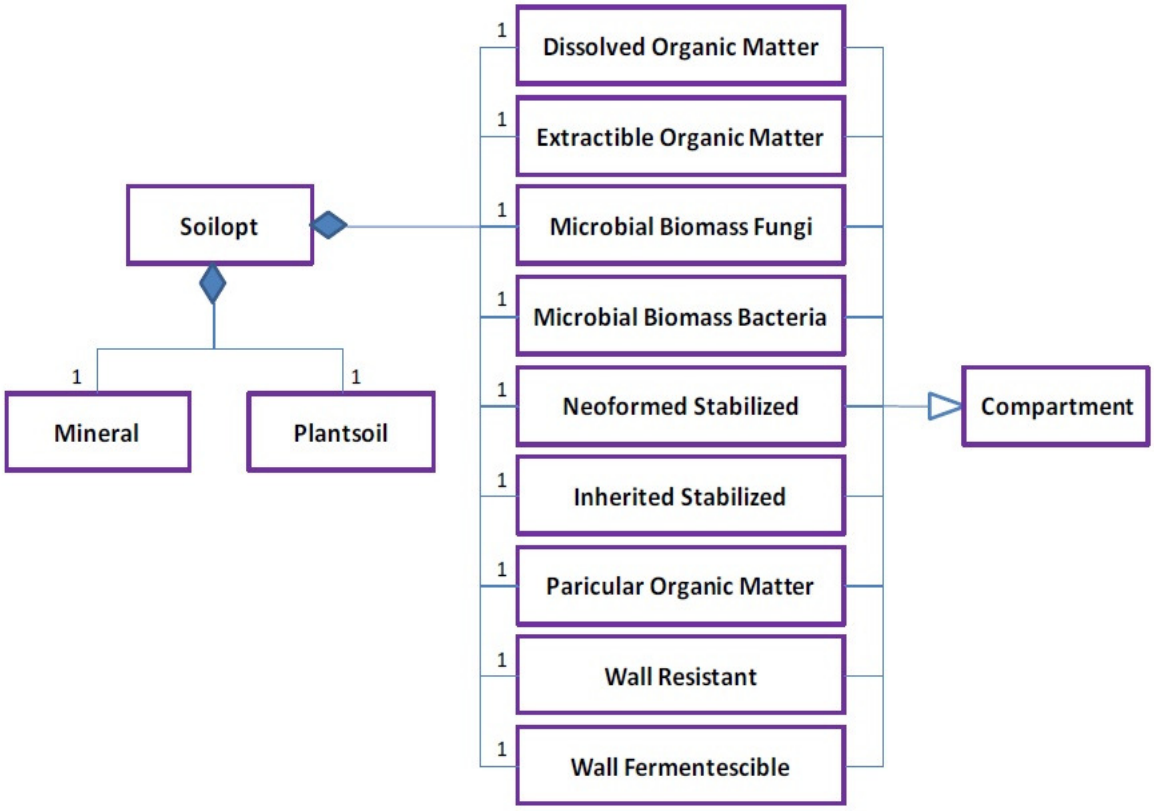


Figure 2 : Diagramme UML simplifié du modèle SOILOPT

- Le modèle CANOPT, créé en 1996 par Jean-François Soussana (INRA), modélise une à plusieurs populations végétales, ces populations étant ou non d'espèces différentes. Ce modèle prend en compte les caractéristiques morphologiques de la plante pour effectuer les calculs relatifs à l'acquisition et à l'utilisation du carbone et de l'azote ainsi qu'à la compétition entre les différentes populations pour ces deux ressources.

- Le modèle GEMINI permet d'utiliser de manière séparée les deux modèles précédents ou bien de les coupler afin de permettre le partage de l'azote du sol entre les microbes et les plantes.

Néanmoins, ce dernier se montrait peu évolutif, à cause principalement de l'architecture de l'application qui rendait difficile l'ajout de nouveaux éléments. En 2004, Stéphane Witzmann, un étudiant de l'ISIMA, a donc réalisé une refonte de GEMINI au cours de son stage de manière à en renforcer la stabilité, l'évolutivité et la portabilité. Il a pour cela créé en C++ une plate-forme de simulation : UNIF (**U**nified **N**umerical **I**ntegration **F**ramework). Comme cela est expliqué dans son rapport ([WITZMANN 2004]) il a conçu l'architecture d'UNIF de manière très générique, afin que celle-ci puisse accueillir n'importe quel modèle à intégrateur. Ce dernier devra en revanche respecter une architecture très précise pour pouvoir fonctionner. UNIF simplifie considérablement l'implémentation d'un modèle puisque un grand nombre de paramètres sont déjà pris en charge au sein de la plate-forme. Dans les faits, GEMINI vient donc se « greffer » sur la plate-forme, tout en conservant ses propres méthodes et propriétés, mais en laissant UNIF gérer leur enchaînement, comme on peut le voir sur le diagramme UML simplifié ci-après :

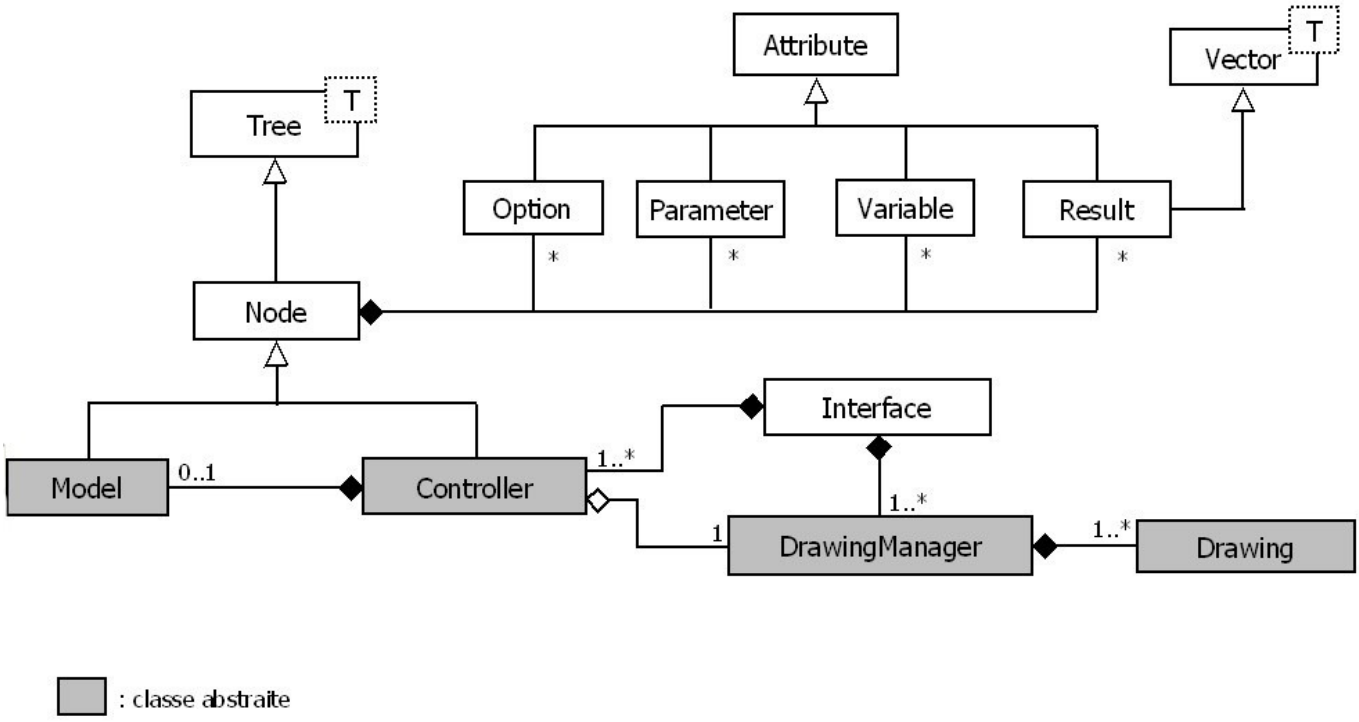


Figure 3 : Diagramme UML simplifié de la plate-forme UNIF

La raison d'être d'UNIF est de fournir aux développeurs de modèles une ossature d'application, permettant à ceux-ci de se concentrer sur l'aspect modélisation plutôt que sur l'aspect simulation. On distingue d'ailleurs très bien cette ossature sur la Figure 3 : la gestion des entités Attributs est effectuée au niveau des *Node* (que nous détaillerons après). L'architecture présente ensuite une couche d'abstraction sous laquelle va venir se « greffer » le nouveau modèle. Ses classes devront donc hériter des superclasses d'UNIF, permettant ainsi la factorisation de bon nombre de méthodes communes à tous les modèles. Pour cela, UNIF propose plusieurs patrons de classe, un simulateur et une interface utilisateur (graphique sous Windows, en ligne de commande sous Unix).

Dans la pratique, pour implémenter un modèle, il suffit donc de spécialiser la classe *Model* (héritage), puisqu'elle contient les caractéristiques génériques à la plupart des modèles de simulation tels qu'une durée de simulation, un pas d'intégration ainsi que les méthodes d'exécution. En effet, une simulation sous UNIF est constituée d'un enchainement de cinq phases, correspondant chacune à une méthode de *Model* :

- *initialize()* permet d'initialiser la simulation (valeurs des variables…).
- *preIntegrate()* permet d'effectuer des calculs avant l'intégration (stockage de la valeur des variables, calculs intermédiaires…).
- *derivate()* calcule la dérivée de chaque variable. L'intégration des variables est ensuite effectuée automatiquement via la méthode d'Euler ou de Runge-Kutta d'ordre 4.
- *postIntegrate()* effectue les opérations postérieures à l'intégration (stockage de résultats…)
- *terminate()* est appelée à la fin de la simulation, elle permet d'achever celle-ci correctement (en vidant les files, en détruisant les objets temporaires…) mais aussi de stocker les données collectées lors de la simulation.

initialize() et *terminate()* ne sont appelées qu'une fois par simulation, respectivement au début et à la fin, tandis que les autres sont appelées cycliquement, dans cet ordre, par la méthode *run()* qui boucle jusqu'à atteindre la date de fin de simulation. D'autres opérations sont bien sûr effectuées

pendant la simulation mais le développeur n'a qu'à écrire ces cinq méthodes spécifiquement pour que son modèle fonctionne, UNIF se chargera de l'enchaînement des actions.

```
void Model::run(bool autoExit, bool doLog)
{
 real_t nextTime;

 cardinal_t step = 0;       // initialize has ever been done

 try {
  reset();

  while (time_ < duration_ and not stop_)         // loop
  {
   nextTime    = (++step)*step_;
   resultTime_ = time_;
   updateAllVariableParameters();

   preIntegrate();            //------------>  first

   firstDerivative_ = true;
   ((*this).*integrate_)();   //------------> second, calls derivate() alone
   time_ = nextTime;

   postIntegrate();          //-------------> third

   while (pause_ and not stop_)  // if the user puts pause
   {}                            // do nothing
  }
 }

 catch (...) {
  clean(doLog);
  throw;
 }
}
```

Figure 4 : Code de la méthode run() de la classe Model

Une autre particularité d'UNIF est d'imposer une architecture arborescente pour ses modèles. On peut le voir facilement sur la Figure 3 avec l'héritage entre *Node* et *Tree*. En effet, *Tree* représente une classe générique d'arbre (utilisation d'un *template*), la classe *Node* hérite donc de *Tree* de type *Node (Tree<Node>)*. En conséquence, un *Node* pourra contenir d'autres *Node* (child). Chaque module doit donc hériter de cette classe, pour pouvoir s'insérer dans l'arborescence. Le *Model* est un *Node* un peu particulier, car il correspond à la racine de l'arbre : ce type de structure permet de créer des modèles comme étant des assemblages de sous-modèles. La relation *Node-Model* pourrait donc être schématisée de la sorte :

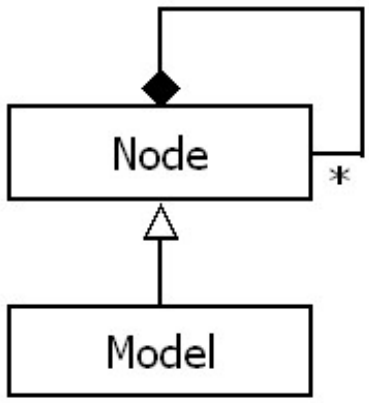


Figure 5 : Relation entre Node et Model

On peut donc voir que la classe *Node* est la classe centrale et principale de l'architecture d'UNIF puisque toutes les propriétés des modèles en découlent : arborescence, liste d'attributs… Mais outre la gestion de la simulation, UNIF fournit également une interface utilisateur très ergonomique et simple qui permet même à un utilisateur non expérimenté d'utiliser la plate-forme.

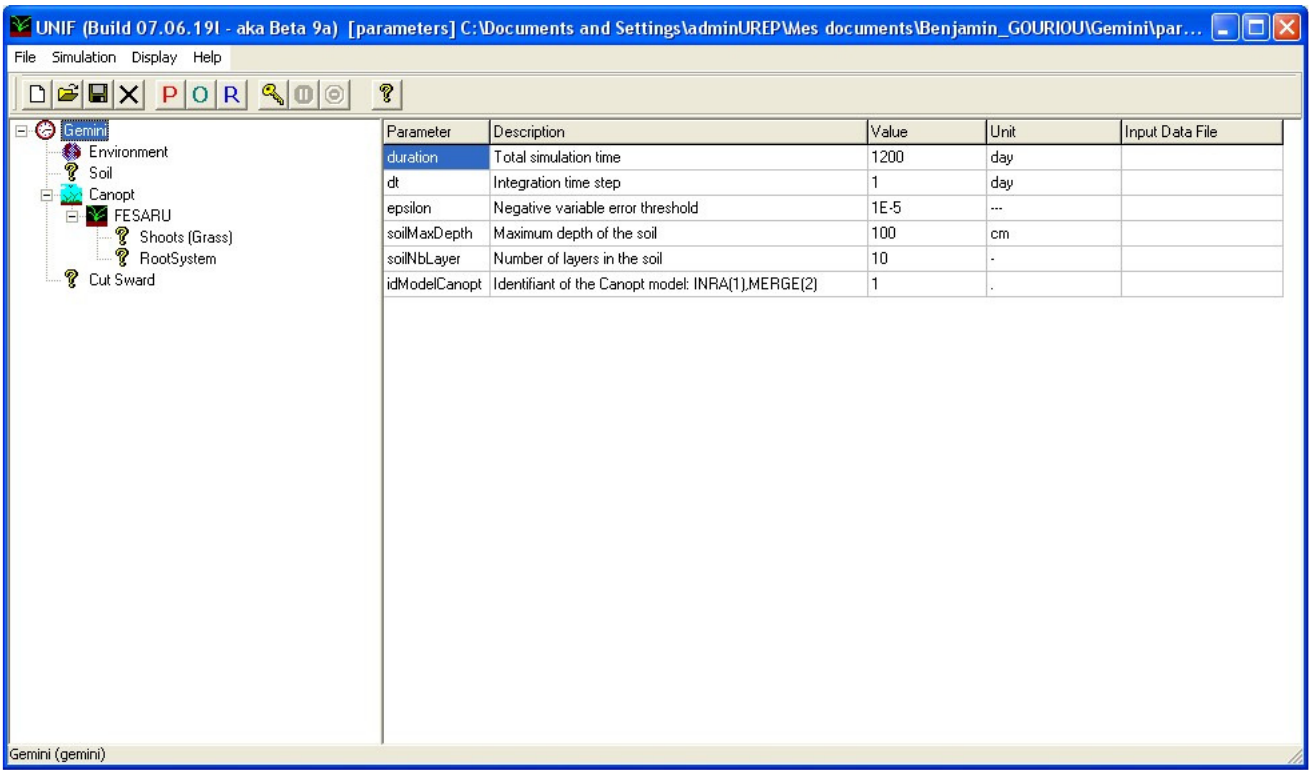


Figure 6 : Interface d'UNIF

Le panneau latéral gauche permet à l'utilisateur de voir rapidement la hiérarchie de ses modèles grâce à l'affichage de leur arborescence. On peut ainsi voir très facilement les différents niveaux de modules et sous-modules.

Le menu supérieur permet d'avoir accès à différentes fonctionnalités :

- créer un nouveau modèle à simuler
- Charger un modèle depuis un fichier de paramétrage
- Enregistrer le modèle courant dans un fichier
- Supprimer le modèle courant
- Accès à la page des paramètres (afin de modifier leur valeur…)
- Accès à la page des options
- Accès à la page des résultats (pour les sélectionner/désélectionner en sortie…)
- Lancer la simulation
- Mettre la simulation en pause
- Stopper la simulation
- Accès à la rubrique « About »

Une fois la simulation terminée, il est possible de visualiser graphiquement les résultats sélectionnés préalablement dans l'interface. En effet, les valeurs de ces derniers sont sauvegardées dans un fichier tableur à chaque pas de temps. Il est donc possible d'ouvrir le fichier directement pour voir les valeurs ou de générer un graphique de ces valeurs en fonction du temps.

1.2. Le modèle GEMINI

GEMINI (**G**rassland **E**cosystem **M**odel with **IN**dividual based **I**nteractions) est un modèle de simulation déterministe, orienté-objet, de flux entre le sol et la plante. Ce modèle a pour but de lier les flux de matières aux traits fonctionnels des plantes afin de comprendre les dynamiques de croissance d'une ou plusieurs populations végétales. Le modèle est centré individu, c'est-à-dire que chaque espèce végétale est représentée par un individu moyen (individu ayant les caractéristiques et le comportement de la moyenne de la population). L'intégration à l'échelle de la population est ensuite effectuée grâce à la densité d'individus. En effet, parallèlement au développement de l'individu moyen, le taux de sénescence et de « naissance » des individus permet de faire évoluer leur densité. Le modèle fonctionne donc toujours à l'échelle d'une population et prend en considération les interactions entre les plantes, avec les éléments du sol, et la compétition pour les différentes ressources (azote, lumière…).

Pour prendre en compte les différentes entités qui sont modélisées et leurs interactions, GEMINI (figure 5) est composé de plusieurs sous-modèles (module) : CANOPT s'occupe de la végétation [MORAND 1999] tandis que SOILOPT gère le sol [BERGIA 1998]. Ces deux sous-modèles sont ensuite couplés avec un module d'environnement, un module de coupe et un module de pâturage, ce qui permet de prendre en compte les modes de gestion des prairies et l'interaction avec les animaux.

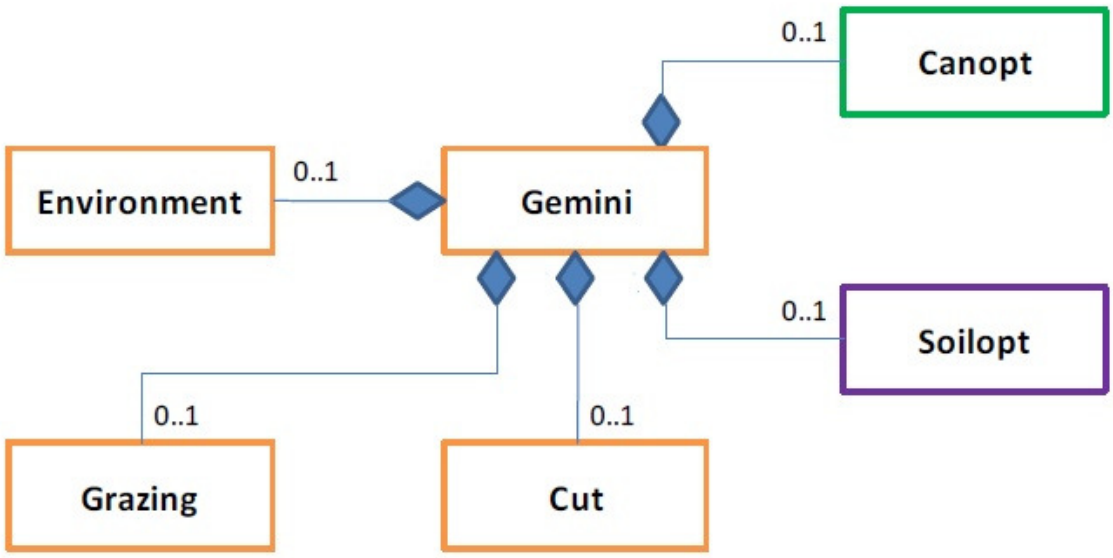


Figure 7 : Diagramme UML simplifié du modèle GEMINI

2. Outils informatiques

Durant mon stage, j'ai utilisé de nombreux outils informatiques, et ce afin de me faciliter la tâche, d'assurer la compatibilité avec les versions précédentes, ou encore d'assurer une qualité suffisante au logiciel produit.

Borland C++ Builder 6 / Codegear C++ Builder 2009

Lors de sa création, la plate-forme UNIF devait fournir à l'utilisateur une interface graphique. Celle-ci a été réalisée avec Borland C++ Builder 6. Cet environnement de développement est un RAD IDE qui comprend un éditeur de code source, un compilateur et un débogueur. Mais surtout, grâce à la bibliothèque de fonctions VCL et à son interface, il propose un outil de création d'interface graphique extrêmement simple et efficace, puisqu'il permet de positionner des éléments graphiques directement sur une fenêtre. Cette simplicité, qui fait la puissance de cet outil, entraîne néanmoins un grand défaut : les interfaces ne sont absolument pas portables, elles dépendent complètement de C++ Builder et ne fonctionnent que sous Windows. Ceci a contraint les développeurs à créer une interface en ligne de commande pour les autres environnements, Linux notamment.

C++ Builder a donc été mon principal outil de travail durant ce stage, puisqu'utilisé en tant qu'IDE lors des phases de codage. Néanmoins, C++ Builder 6 est un outil vieillissant (2002). Son ergonomie, mais aussi ses performances sont aujourd'hui dépassées. Il a donc été décidé de porter GEMINI sur une version plus récente de cet IDE : Codegear C++ Builder 2009. Ce choix a entrainé de grosses modifications du code, en raison du changement de plusieurs fonctions des bibliothèques du programme. Une fois cela réalisé, les qualités du nouveau compilateur ont permis d'accroître la stabilité du programme.

Subversion / TortoiseSVN

Subversion (SVN) est un logiciel de gestion de versions open-source qui s'appuie sur un dépôt de données centralisé et unique. Il permet à une équipe de travailler sur un projet en ayant accès en permanence à l'ensemble des versions des sources, y compris bien sûr la dernière. Chacun est libre de « valider » les modifications qu'il a apportées afin que ces dernières soient remontées au niveau du serveur, et donc disponibles aux autres membres de l'équipe. On peut aussi restaurer une version antérieure en cas de problème, rechercher une modification ou même visualiser l'arborescence des versions. SVN est mis en place sur le serveur de l'INRA, et donc tous les postes peuvent avoir accès au dépôt des données, avec bien sûr des restrictions sur les accès utilisateurs.

TortoiseSVN est un logiciel client qui propose une interface graphique à SVN. Il s'intègre à l'explorateur de fichiers Windows. Il suffit donc par exemple de faire un clic droit sur le dossier du projet pour avoir le menu SVN. De plus, les icônes sont modifiées afin de voir instantanément si le fichier est à jour ou non…

Autres outils utilisés

De nombreux autres outils ont été utilisés durant le stage, je ne les présenterai pas tous mais je vais citer les plus importants :

- Silverpeas est une plate-forme collaborative intégrée développée en Java qui permet de gérer un portail intranet dans une entreprise ou au sein d'une équipe. Cette plateforme permet par exemple de tenir à jour un agenda, de gérer des projets mais aussi et surtout de publier et versionner des documents et articles qui seront accessibles aux autres utilisateurs ayant les autorisations.

- Tout projet informatique se doit d'avoir une documentation complète et claire. Pour le projet GEMINI, c'est Doxygen que nous avons utilisé afin de générer automatiquement de la documentation à partir du code source du programme. A partir de commentaires formatés de manière spécifique (type javadoc) et du code source, Doxygen est capable de fournir une documentation mais aussi des diagrammes de classe avec description ainsi que des graphes d'appels (avec l'aide du logiciel graphviz). La documentation générée peut être sous plusieurs formats, dans notre cas il s'agit de l'HTML.

- Enfin, WinMERGE m'a été très utile pour la première partie de mon stage. Ce logiciel Open Source est un outil de comparaison et de fusion de fichiers. Il permet de déterminer les différences entre deux fichiers, et les présente visuellement dans un format texte qui est facile à comprendre et à manipuler.

Une fois muni de ces outils, et après avoir étudié l'environnement de travail, je me suis attaché à mieux comprendre quels étaient les objectifs de mon stage. J'ai pu déterminer plusieurs étapes, me permettant ainsi d'établir un échéancier précis sur le travail à effectuer. La priorité me semblait être la réalisation de la nouvelle version regroupant les anciennes, permettant de « synchroniser » les travaux de l'UREP avec ceux de l'équipe du Max Planck Institute avant d'ajouter d'autres fonctionnalités comme l'eau. Je me suis donc lancé dans cette tâche.

Réalisation d'une version hybride des modèles existants

1. Problématique

Il existe 3 versions du modèle GEMINI :

- une version développée par l'UREP, qui est aussi la première version du modèle, représentant l'architecture des plantes de manière simplifiée (hyperbole ou bâtonnet pour les feuilles et arbre pour les racines).
- une version développée par une équipe de recherche du Max Planck Institute à JENA en Allemagne. En parallèle de la première version, cette équipe développe une architecture des plantes plus complexes permettant d'inclure dans le modèle GEMINI une plus grande diversité des plantes dont notamment les dicotylédones non légumineuses. Seule l'architecture des plantes est modifiée, la majeure partie du programme reste identique à la version UREP.
- enfin la dernière version, dite « MERGE », est la version résultant de la fusion des deux versions précédentes, qui reprend donc les propriétés de ces dernières, et est destinée à devenir la version principale sur laquelle les modifications futures seront développées.

L'objectif est donc d'implémenter un unique modèle permettant de sélectionner la version à utiliser. L'intérêt est que la majorité des fonctions sont factorisables, aussi les modifications/corrections qui seront apportées dans le futur seront de facto impactées sur les différentes versions. De plus, l'objectif à moyen terme est d'utiliser une unique version, réunissant l'ensemble des développements effectués sur les deux versions. Cet objectif sera facilement atteint une fois ce modèle hybride réalisé.

2. Analyse de l'environnement de développement : Canopt

La version développée par l'équipe de JENA étant dérivée de la version INRA, les modifications de code à effectuer afin de réaliser la version mergée sont peu nombreuses, et surtout seront facilement décelables via une comparaison des fichiers deux à deux. Il a ainsi pu être établi que seule la modélisation de la morphologie de la plante était différente entre les deux versions, aussi l'essentiel du travail se portera sur le modèle de végétation, Canopt.

Ce modèle – CANopy OPTimisation - décrit la morphogénèse aérienne et racinaire d'une ou plusieurs populations végétales, ainsi que les compétitions pour la lumière (partie aérienne) et pour l'azote minéral du sol (partie racinaire). Une plante (ou population) peut appartenir à trois familles différentes : les graminées (grass), les légumineuses (legume) ou les herbes « diverses » (herb) qui sont non-fixatrices d'azote. Les deux parties de la plante (racine et feuille) ainsi que les différentes familles sont modélisées afin de pouvoir leur associer des caractéristiques propres.

Toutefois, suivant la version, la modélisation de la plante n'est pas exactement la même. Ainsi, la version développée par l'UREP correspond au modèle de base et comprend donc les

caractéristiques décrites ci-dessus. Comme on peut le voir sur son diagramme UML simplifié (Figure 8) les modèles peuvent comporter plusieurs espèces de plantes (*Plantcan*) appartenant à une des trois familles qui la spécialisent. Chacune de ses plantes (individu moyen comme expliqué précédemment) possède une partie aérienne (*Shoots*) et une partie racinaire (*RootSystem*).

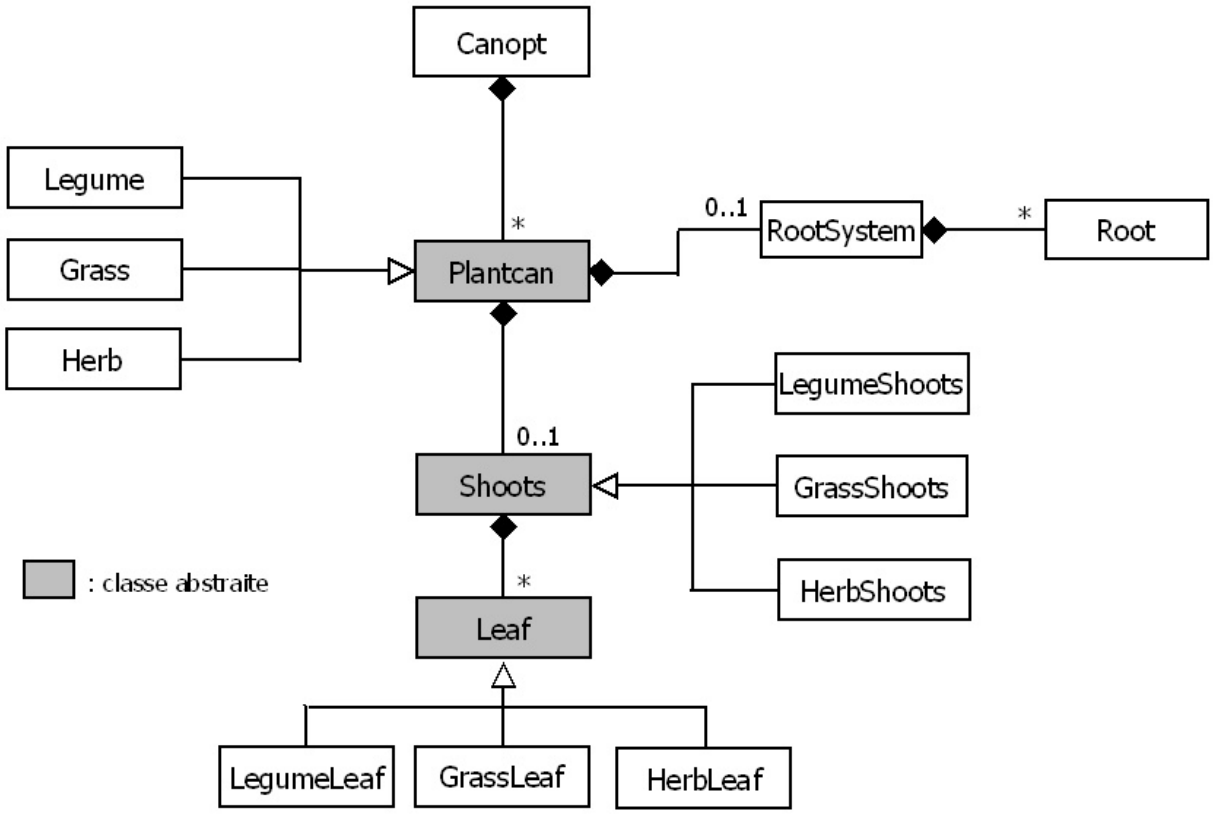


Figure 8 : Diagramme UML simplifié Canopt INRA

Dans la version développée à JENA, quelques spécifications supplémentaires ont été ajoutées, en particulier la modélisation des phytomères (branch en anglais). En effet, dans la version de base, GEMINI ne prend pas en compte la tige ni les gaines des feuilles, bien que ces éléments modifient l'élévation de la plante et donc la façon dont elle absorbe la lumière. L'équipe de l'institut Max Planck a donc fait évoluer l'architecture des parties aériennes vers cette structure arborescente des feuilles (*children*).

Enfin, la version MERGE correspond à une fusion des versions INRA et JENA, c'est donc fort logiquement qu'elle hérite des deux autres et implémente ensuite ses propres caractéristiques. Néanmoins l'architecture même du modèle ne diffère pas de la version JENA. Les deux versions ont le même diagramme de classe sur lequel apparait clairement la différence de modélisation par rapport à la version UREP : figure 9.

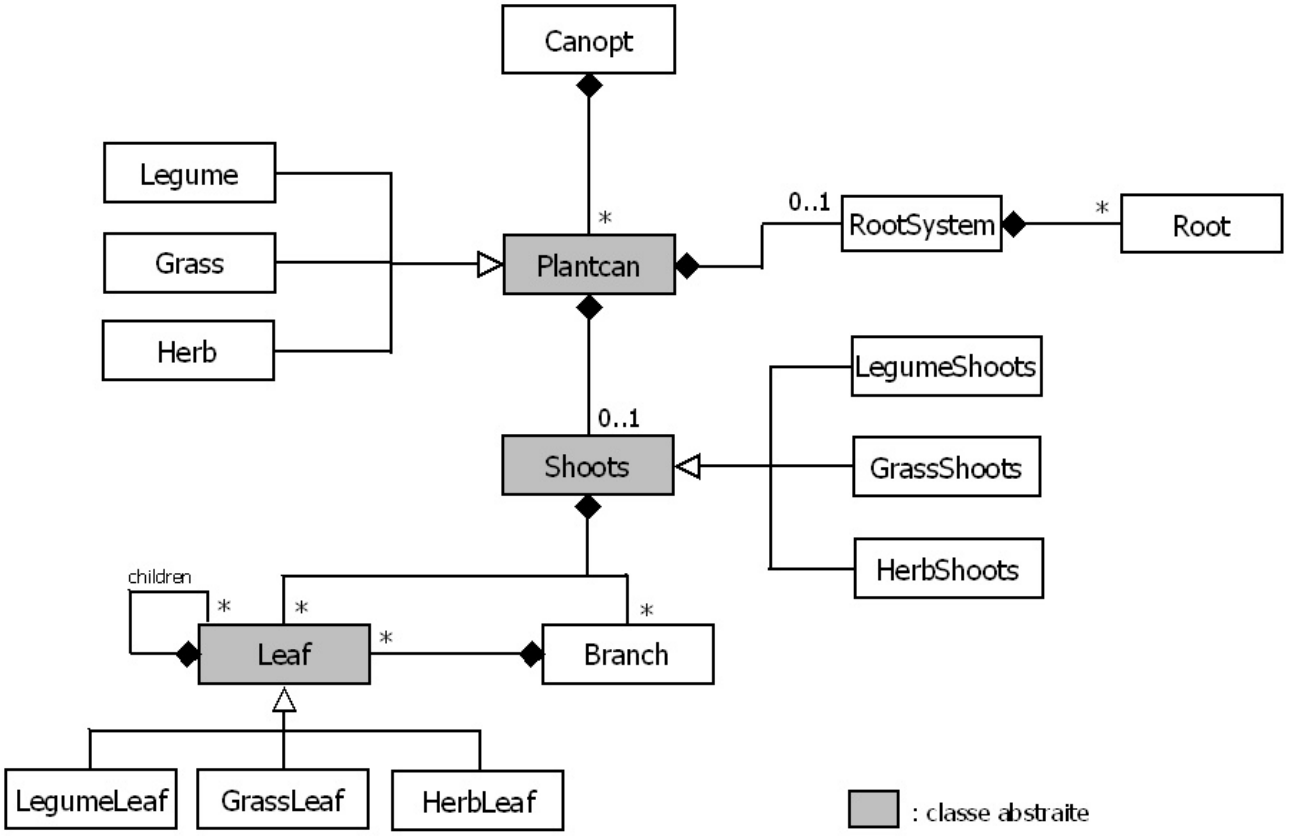


Figure 9 : Diagramme UML simplifié Canopt MERGE / JENA

Après analyse, la majeure partie des modifications se situeraient dans les classes « *Shoots* », « *Leaf* » et « *Branch* », cette dernière n'existant que dans la version MERGE. Hormis quelques ajouts ou différences de code mineurs, les autres classes demeureraient identiques. Nous avons donc décidé d'implémenter le « *switch* » au niveau de ces classes : modifier leur implémentation pour intégrer les deux versions du modèle, puis instaurer un mécanisme de choix du modèle à utiliser.

3. Etude des différentes possibilités d'implémentation

La méthode de choix du modèle la plus simple et la plus naturelle est l'ajout d'un paramètre. Il doit bien sûr être disponible dans l'interface, permettant ainsi à l'utilisateur de choisir quelle version il souhaite utiliser (1 pour INRA, 2 pour MERGE) Ce paramètre est un entier, placé dans la classe *Gemini* de façon à être accessible au plus tôt à l'utilisateur. La première fenêtre à s'afficher lors du lancement de l'interface graphique permet alors de renseigner sa valeur.

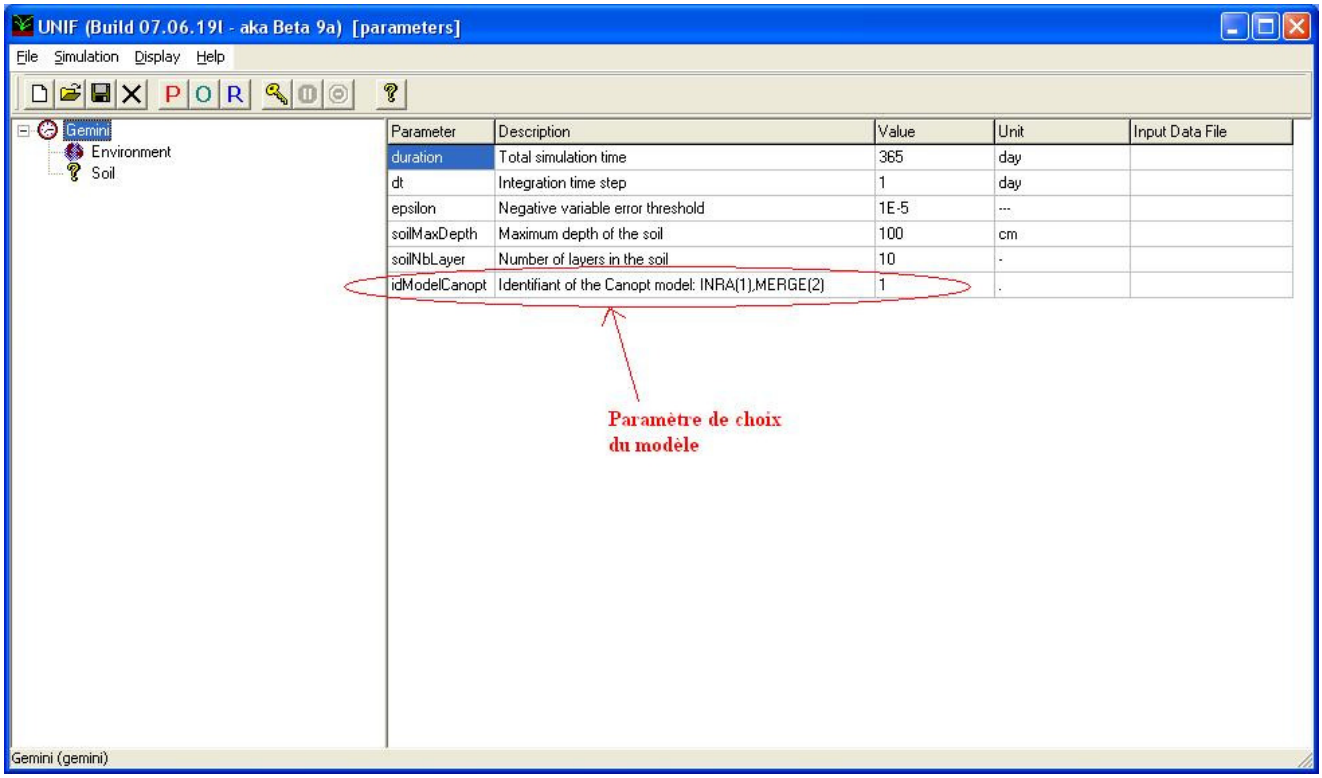


Figure 10 : Paramètre de choix du modèle dans l'interface

Une fois le mécanisme de choix sélectionné, la seconde étape fut de s'intéresser à l'implémentation des classes, tout en gardant à l'esprit que le modèle allait encore subir des modifications. De plus, il fallait mutualiser un maximum de propriétés des différentes versions. Afin de réaliser cela, plusieurs solutions étaient envisageables.

3.1. L'utilisation d'un héritage conditionnel

La première solution envisagée fut de regrouper la partie commune des trois versions dans une superclasse abstraite. Ensuite des classes filles hériteraient de cette superclasse, celles-ci comprenant les données propres à chaque modèle. Ceci est résumé sur le graphe UML simplifié suivant :

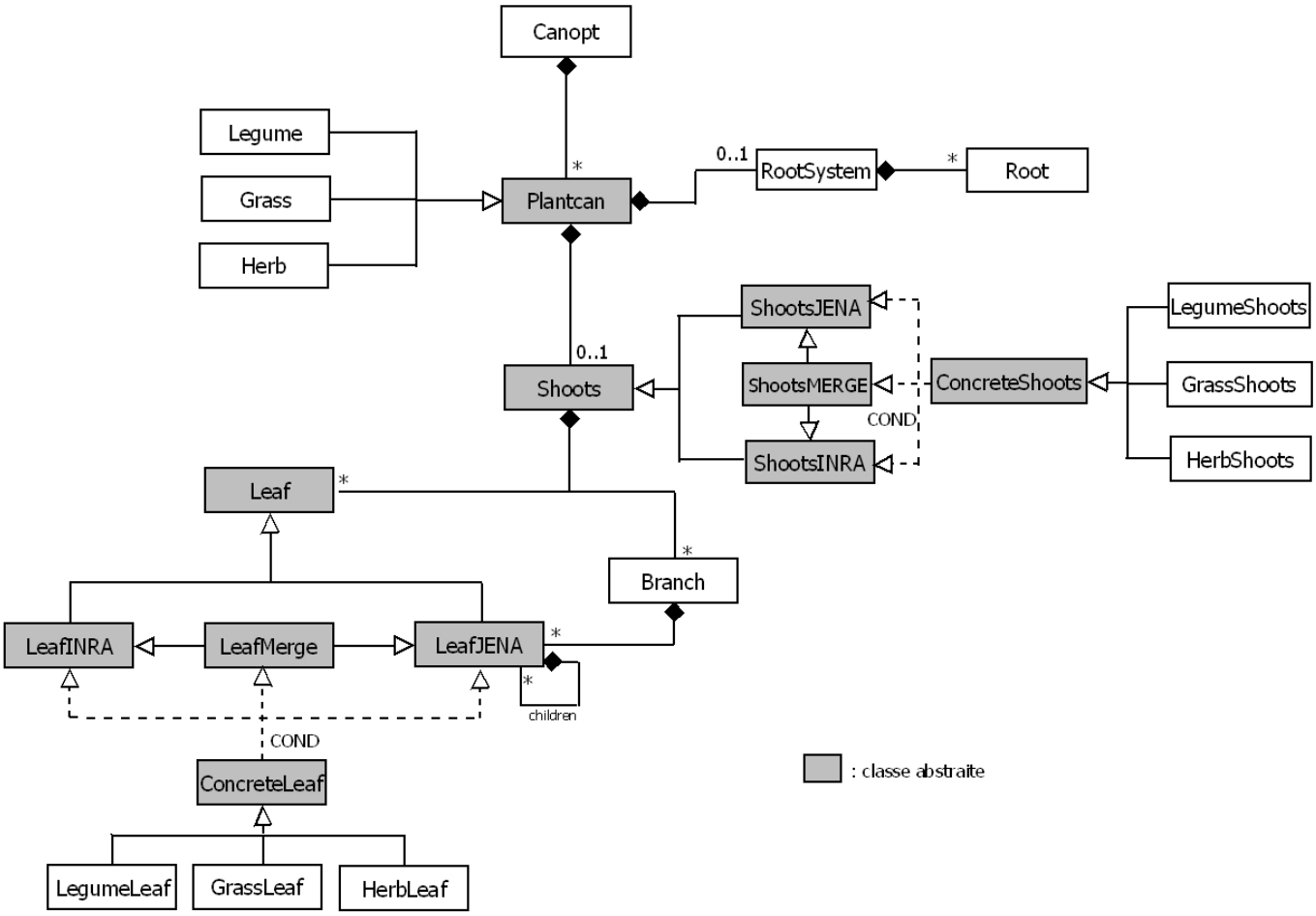


Figure 11 : Solution 1 - Héritage conditionnel

Ce diagramme montre très bien l'objectif de la méthode : les différentes versions sont implémentées dans les classes correspondantes (*ShootsINRA*, *ShootsJena* et *ShootsMerge*) et toutes dérivent de la superclasse Shoots puisqu'elles ont toutes des caractéristiques communes (idem pour *Leaf*). On peut voir de plus que *ShootsMERGE* hérite elle-même des deux autres versions. C'est la façon la plus simple et la plus intuitive de mettre en évidence les relations entre les versions du programme. Néanmoins, les nombreuses interactions avec l'équipe allemande Jena nous ont décidé à ne pas intégrer au programme la version développée à Jena. En effet, cette version est très similaire à la version MERGE et ne sera plus utilisée dans le futur. Nous ne détaillerons donc pas cette version bien que certains schémas en fassent référence, il était en effet prévu de l'intégrer au début de mon travail.

Une autre classe, *ConcreteShoots*, a été ajoutée. Cette classe abstraite sert de passerelle afin d'utiliser un héritage conditionnel. Elle permet, grâce à une structure *Selector*, d'hériter de la bonne classe suivant la valeur du paramètre de choix. Comme on peut le voir dans l'extrait de code source (figure 11), les classes *GrassShoots*, *LegumeShoots* et *HerbShoots* hériteront de la classe *ConcreteShoots* à laquelle on passera en argument de *template* la valeur du paramètre de choix (entre 1 et 3). Ce paramètre sera utilisé ensuite par le Selector pour déterminer le type correspondant : « INRA » ou « MERGE ». En effet *ConcreteShoots* hérite elle du type défini par le *Selector* (suivant la valeur de l'entier en paramètre), donc soit *ShootsINRA* soit *ShootsMERGE*. Ainsi, ces classes héritent toujours de *Shoots* avec le nouveau modèle, en spécialisant l'une des versions du modèle. Voici un extrait de code de démonstration du fonctionnement de *Selector*.

```
template<int G, typename T, typename U, typename V>
struct Selector
{
    typedef T type;
};


template<typename T, typename U, typename V>
struct Selector<2, T, U, V>
{
    typedef U type;
};


template<typename T, typename U, typename V>
struct Selector<3, T, U, V>
{
    typedef V type;
};


// C o n c r e t e S h o o t s  Interface //---------------------------------

template <int G>
class ConcreteShoots : public Selector<G, ShootsINRA, ShootsMERGE, ShootsJENA>::type
{

 //-------------------------------------------------------------------Constants

  typedef Selector<G, ShootsINRA, ShootsMERGE, ShootsJENA>::type typeshoots;

 //-------------------------------------------------------------------Constructor
  public: Shoots(Plantcan & _plant,cNodeID & _id,cString & _name) : typeshoots(_plant, _id, _name) {}

};
```

Figure 12 : Extrait du code de l'héritage conditionnel, fonctionnement de Selector

Problème rencontré : Cette méthode a l'avantage d'être extrêmement simple à mettre en œuvre, le choix entre les différents héritages étant géré tout simplement par le *Selector*. Malheureusement, avec cet héritage « par *template* », le type utilisé pour l'héritage de la classe *ConcreteShoots* est défini à la compilation, ce sera la valeur d'initialisation du paramètre. Il est ensuite impossible de le modifier par la suite. Ainsi, à moins de compiler le programme avant chaque exécution, cette solution n'était pas applicable ici. Néanmoins cette technique reste très utile pour effectuer un héritage conditionnel tant que la condition n'est pas rentrée en paramètre au programme. Par exemple le code précédent pourrait servir si on avait dans une autre méthode ou dans la fonction *main() :*

new ConcreteShoots<1>() ;
new ConcreteShoots<2>() ;

En effet, ici la création et le paramètre *template* sont écrits en « dur » contrairement à notre cas où la valeur du paramètre n'est fixée qu'au lancement du programme et pas à la compilation.

3.2. Utilisation de *template* pour les spécialisations

Cette solution est très proche de l'héritage conditionnel, à ceci près que le problème lié à la valeur du paramètre à la compilation est surmonté. En effet, ici on rend paramétrables toutes les classes spécialisées de *Shoots* et *Leaf* en les mettant sous forme de classe *template*. Celles-ci héritent donc directement de la classe que l'on passe en paramètre du *template*.

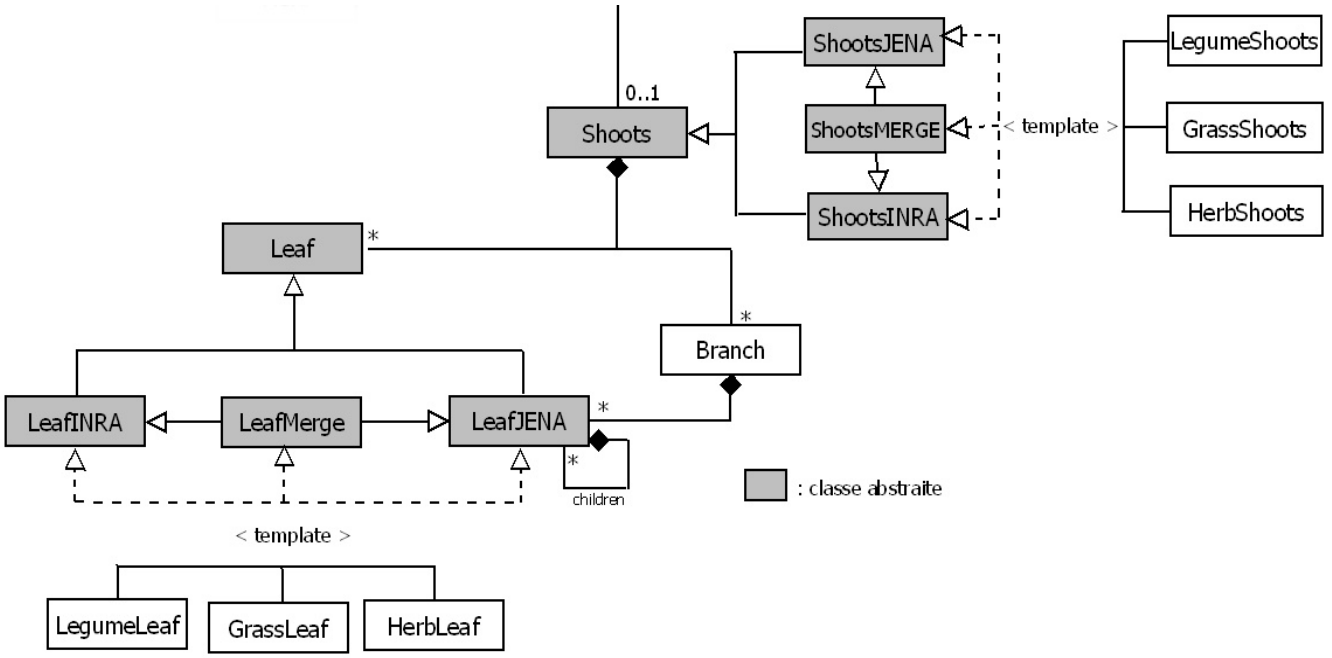


Figure 13 : Diagramme UML simplifié de la solution par "template"

Le problème précédent est donc contourné à la création des classes. On crée dynamiquement ces objets dans les méthodes des classes *Grass, Legume* et *Herb* correspondantes. A la création, on réalise alors un « switch » sur la valeur du paramètre, et dans chaque cas on crée la classe en passant en paramètre le type correspondant. L'extrait du code source ci-dessous permet d'illustrer le fonctionnement de cette généricité et de la création dynamique des instances :

```
//-------------------------Style de déclaration des classes

template <class T>
class GrassShoots : public T
{
    GrassShoots() : T() {}
    //...
};


//---------------------------Création des instances

switch (canopt ::IDmodel) {
        case 1 : {
                        _ shoots = new GrassShoots<ShootsMERGE>(*this);
                        break;
                }
        case 2 : {
                        _ shoots = new GrassShoots<ShootsINRA>(*this);
                        break;
                }
        case 3 : {
                        _ shoots = new GrassShoots<ShootsJENA>(*this);
                        break;
                }
    }
```

Figure 14 : Extrait du code de l'utilisation des « template » pour l'héritage

Problème rencontré :

Cette méthode permet de s'affranchir des problèmes posés par la méthode de l'héritage conditionnel. Néanmoins ce système d'héritage implique l'appel systématique du constructeur par défaut de la classe T (paramètre *template*). Or les classes *ShootsX*, où X est le nom de la version, ne disposent pas de tels constructeurs puisqu'elles ont des attributs qui doivent impérativement être initialisés, et appellent de plus le constructeur de Shoots qui lui-même appelle celui de *Node*. Cette dernière classe comporte des attributs référence imposés par la plateforme UNIF. Cette solution n'est donc pas utilisable ici puisqu'elle ne répond pas aux critères d'intégration à la plateforme même si dans un autre cas elle serait applicable en modifiant certains attributs des classes.

Nous avons donc cherché une autre solution en voyant le problème sous un angle différent.

3.3. Utilisation de l'agrégation à la place de l'héritage

Nous avons cherché une solution qui permettrait d'éviter tous les problèmes soulevés par les précédentes méthodes. Sous les conseils de Monsieur David Hill, nous avons donc opté pour l'utilisation de l'agrégation en lieu et place de l'héritage. En d'autres termes, « une pousse d'herbe est une spécialisation de la pousse » devient « une pousse d'herbe possède une pousse », ce qui reste tout a fait correct au niveau de la modélisation. D'ailleurs, la plupart des cas d'héritage peuvent être remodelés sous forme d'agrégation. Cette technique permet d'éviter d'utiliser des *template* dans toutes les classes spécialisées et surtout contourne tous les problèmes liés à l'héritage. De plus, cela donne la possibilité d'utiliser un patron de création : le pattern « Abstract Factory ». Néanmoins un remaniement général du code est nécessaire : tous les paramètres hérités accessibles directement (de façon transparente) par héritage doivent maintenant être référencés dans les classes voulant les utiliser, car le principe d'encapsulation empêche de laisser l'ensemble des attributs en accès public.

3.3.1. Le patron Abstract Factory

La *Factory* est un patron de conception de création utilisé en programmation orientée objet. La « fabrique » a pour rôle l'instanciation d'objets divers dont le type n'est pas prédéfini : les objets sont créés dynamiquement en fonction des paramètres passés à la fabrique. Comme en général les fabriques sont uniques dans un programme, on utilise souvent le patron de conception singleton pour les implémenter.

Le patron Abstract Factory permet quant à lui de rajouter une couche d'abstraction au patron Factory. L'*AbstractFactory* instancie une fabrique spécialisée pour une famille d'objets. Contrairement au patron Factory qui permet d'instancier un objet d'une classe donnée, *l'Abstract Factory* permet donc d'instancier un objet parmi un choix de plusieurs classes différentes, à chacune d'entre elles correspondant une Factory spécialisée.

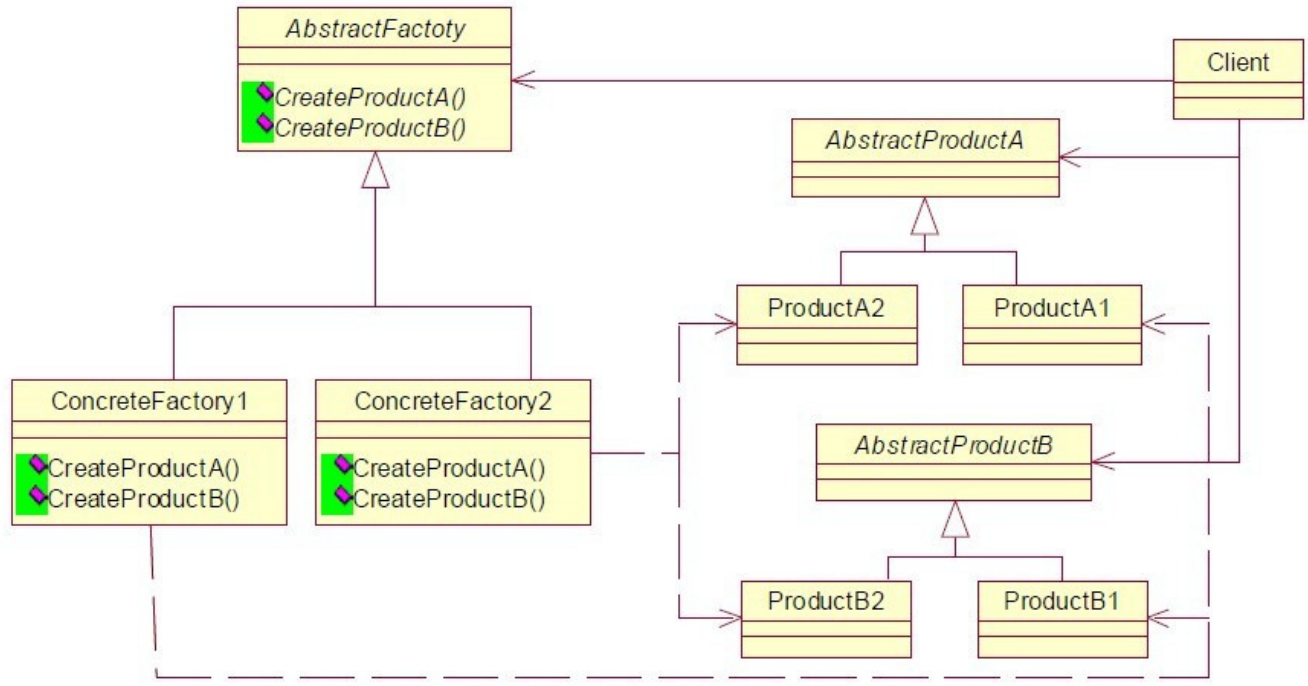


Figure 15 : Diagramme UML générique du patron Abstract Factory

3.3.2. L'application du patron sur le modèle

En utilisant le patron Abstract Factory, on contrôle la création des pousses (shoots en anglais) afin qu'elles soient du type voulu. En effet, on va implémenter une *Factory* pour chaque type de pousse souhaité : *ShootsINRA_Factory*, *ShootsMERGE_Factory*. Chacune de ces classes sera bien sûr en charge de l'instanciation des pousses qui lui correspondent grâce à la méthode héritée de la classe *Shoots_Factory*. Il suffit donc, à l'initialisation, de créer la Factory qui correspond à la version du modèle choisi. Une fois ceci effectué, il n'est plus nécessaire de faire un test sur la valeur du paramètre de choix de la version à chaque fois que l'on crée une pousse, il suffit de faire appel à l'*AbstractFactory* qui, par polymorphisme d'héritage, instanciera le bon type de Shoots automatiquement.

Voici un diagramme UML simplifié de Canopt après l'application de cette méthode :

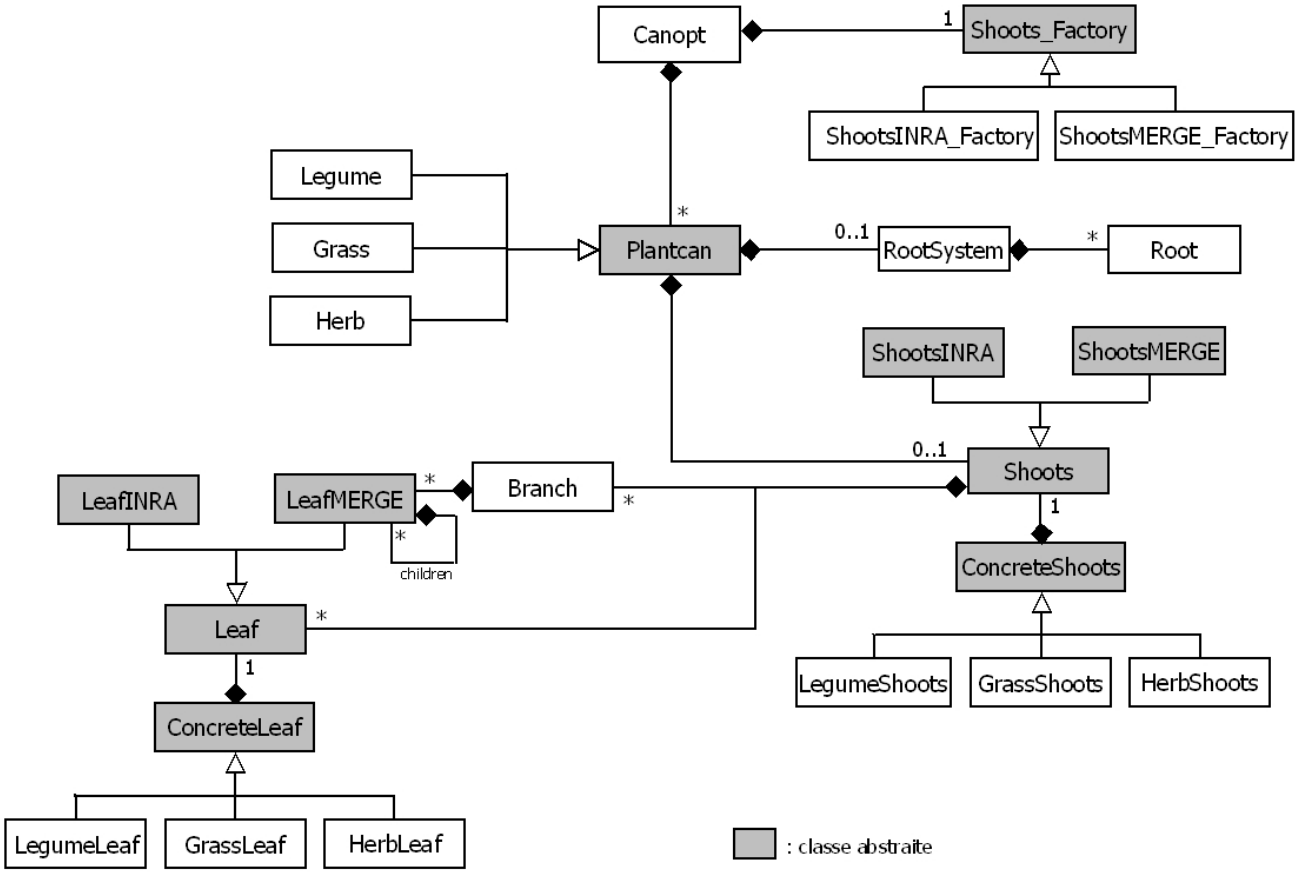


Figure 16 : Diagramme UML simplifié de la version finale

Outre l'ajout des *Factory*, on peut voir sur le diagramme, au niveau des classes *ConcreteLeaf* et *ConcreteShoots*, que l'héritage a fait place à l'agrégation. Il est possible, via son adresse, d'accéder aux attributs et méthodes publiques d'un objet. Néanmoins, avec le principe d'encapsulation les attributs de classe doivent être déclarés « protected » ou « private ». L'héritage permet aux classes dérivées d'accéder directement aux membres « protected » de leur classe mère mais pas l'agrégation. Néanmoins la notion de classe amie permet de palier à ce problème. En effet en déclarant une classe amie d'une autre, les attributs (« public » et « protected ») de cette dernière lui deviendront accessibles. C'est pourquoi les classes de version comme (*LeafINRA* et *LeafMERGE*) sont amies des

classes de famille (*GrassLeaf*, *LegumeLeaf* et *HerbLeaf*), et ce afin de pouvoir accéder à toutes les informations nécessaires aux calculs.

On peut remarquer qu'il n'existe pas de Leaf *Factory*. En effet, *Leaf* étant créée à partir de la classe *Shoots*, il n'est pas nécessaire d'avoir une différenciation à ce niveau là. Chaque *Shoots* créera la *Leaf* qui correspond à sa version du modèle. Voici une partie du code qui permet de mieux comprendre les mécanismes de cette méthode.

```
// S h o o t s _ F A C T O R Y  Interface //-------------------------------------------------------------------
/** Represents the abstract Factory to build a specific factory (INRA or MERGE) */
class Shoots_Factory
{
    public: virtual Shoots * createShoots(Plantcan &,cNodeID &,cString &,ConcreteShoots *) = 0;

    protected: static Shoots_Factory * InstFacSh;            //< Ptr to a specific factory (type of shoots)
    public: static Shoots_Factory * InstanceFactoryShoots(); //< Create the specific factory
};

// S h o o t s _ F A C T O R Y  Implementation //-------------------------------------------------------------
Shoots_Factory * Shoots_Factory::InstFacSh=NULL;

Shoots_Factory * Shoots_Factory::InstanceFactoryShoots()
{
   if (InstFacSh==NULL)
   {
      if (Canopt::IDMODEL == 1)
         InstFacSh = new ShootsINRA_Factory();
      else
         InstFacSh = new ShootsMERGE_Factory();
   }

   return InstFacSh;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////

// S h o o t s I N R A _ F A C T O R Y  Interface //-----------------------------------------------------
/** Use to create a shoot of the type INRA */
class ShootsINRA_Factory : public Shoots_Factory
{
      public: Shoots * createShoots(Plantcan &,cNodeID &,cString &,ConcreteShoots *);
};

// S h o o t s I N R A _ F A C T O R Y  Implementation //-----------------------------------------------
/** Use to build a ShootsINRA because the constructor is protected (friend class) */
Shoots * ShootsINRA_Factory::createShoots(Plantcan & _pl,cNodeID & _id,cString & _name,ConcreteShoots * _conc)
{
      return (new ShootsINRA(_pl,_id,_name,_conc));
}
```

Figure 17 : Extrait du code relatif au mécanisme "Factory"

On peut voir dans ce code que les classes *Shoots_Factory* et *ShootsINRA_Factory* sont implémentées de manière à être unique. En effet, dans la première, un pointeur et une méthode statiques permettent de référencer et d'instancier une *Factory* spécialisée (*ShootsMERGE_Factory* ou *ShootsINRA_Factory*) par héritage. Puisqu'elles sont statiques, la méthode est accessible sans qu'aucune instance de l'objet n'ait été créée. La méthode *InstanceFactoryShoots()*, elle, renverra

l'adresse de la *Factory* spécialisée qui existe déjà et s'il n'en existe pas encore, en créera une nouvelle d'après la valeur du paramètre de choix du modèle. C'est le principe du Singleton : un patron de classe qui assure l'unicité et la facilité d'accès de la classe qui respecte cette architecture. Ainsi, lors du premier appel de cette méthode, une *Factory* sera créée, du type correspondant à la valeur du paramètre de choix, puis, lors des appels suivants, cette méthode renverra l'adresse de la Factory préalablement créée. On peut noter que les constructeurs des classes *ShootsMERGE* et *ShootsINRA* sont « protected », ils ne sont donc accessibles qu'aux classes filles ou aux classes amies, qui sont bien entendu leurs *Factory* respectives. De cette façon, on est sûr que seule la Factory adéquat pourra créer les *ShootsMERGE* ou *ShootsINRA* correspondante. De plus, un autre intérêt, induit par le fait que la Factory est unique, est qu'il ne peut donc pas y avoir de mélange de versions : toutes les pousses créées seront soit du type INRA soit du type MERGE. C'est donc une sécurité pour éviter un éventuel mélange des versions qui n'aurait aucun sens. Enfin, la création d'une instance est grandement facilitée puisque les *Factory* sont accessibles de partout grâce à leurs attributs statiques :

Shoots_Factory::InstanceFactoryShoots() -> createShoots(// "parametres"…);

Au regard de l'étude de ces différentes possibilités, cette solution de remplacer l'héritage par de l'agrégation nous a semblé être la meilleure puisqu'elle répondait aux contraintes de la plateforme et à l'objectif qui nous était fixé. Nous l'avons donc implémentée puis testée afin de contrôler la non-régression du modèle.

Développements du modèle GEMINI

1. Développements annexes

1.1. Génération et chargement de fichiers de paramétrage

L'étape de test m'a été grandement facilitée par un outil développé précédemment par Raphaël Martin : un programme de génération de fichiers de paramétrage, écrit en C++ sous Linux, permettant de créer des mixtures (mélange) d'un nombre quelconque de plantes. Les plantes contenues dans la mixture sont générées aléatoirement à partir de fichiers textes contenant une plante paramétrée par Vincent Maire au cours de sa thèse. Les plantes sont ainsi intégrées au fichier de paramétrage, en respectant son architecture. En effet ce dernier est composé de l'enchainement des modules que l'on souhaite utiliser dans le modèle dans leur ordre hiérarchique avec, au sein de chacun d'eux, différents attributs : paramètres et valeur d'initialisation, résultats produits en sorties et un booléen (1 ou 0) qui précise s'ils doivent être générés ou non, ou options et valeur d'activation (« Yes » ou « No »). Vous trouverez en Annexe 1 le début d'un fichier de paramétrage ainsi que les explications associées permettant de mieux comprendre la disposition des données et donc le système de chargement.

Il a fallu, dans un premier temps, modifier le programme afin qu'il génère des fichiers interprétables par le modèle hybride. Pour cela, le paramètre choix du modèle a été ajouté au sein des fichiers de paramétrage produits ainsi qu'un nouveau résultat qui devait apparaître en sortie. Cela effectué, les fichiers d'entrée fonctionnaient très bien pour chacune des versions, ils ne pouvaient pas être intervertis (un fichier de la version UREP n'était pas utilisable avec la version MERGE et vice-versa). Or pour comparer les résultats des deux versions dans les mêmes conditions, utiliser les mêmes fichiers diminuerait le risque d'erreur. Il suffirait alors juste de changer la valeur du paramètre de choix du modèle pour relancer directement à la suite une autre simulation. Mais la fonction de chargement d'UNIX possède une gestion des exceptions très complète qui contrôle lors de la lecture dans le fichier si l'attribut est bien présent dans le module. Si ce n'est pas le cas UNIX lance une exception et affiche le nom de l'attribut qui n'est pas dans le modèle. Pour pouvoir mettre dans le même fichier les attributs des deux modèles sachant qu'un seul n'est utilisé à chaque simulation, il faut donc contourner la gestion des exceptions mais sans la mettre à bas puisqu'elle reste très utile en cas de faute de frappe par exemple.

Ce problème se révèle particulièrement ardu, puisque contrairement au Java, en C++ il n'y a pas d'accès possible aux membres de classe lorsque celle-ci n'est pas instanciée, hormis les membres *static*. Par défaut nous devons donc nous servir d'attributs *static*. Une seule solution est apparue : utiliser un tableau *static* dans chaque classe qui contient les « Attributes » de cette classe pour les référencer. Ainsi à chaque fois qu'un utilisateur désire rajouter un Attribute il doit ajouter dans le tableau *static* de la classe le nom de ceui-ci. Grâce à cela même si la classe n'est pas instanciée lors de l'exécution, il est possible, en cherchant dans le tableau, de savoir ce que contient cette classe (en tout cas les paramètres, options et résultats). Une méthode *static* a donc aussi été définie pour contrôler (dans le tableau) si un Attribute donné est ou non dans la classe, comme on peut le voir dans l'extrait de code ci-dessous :

```
//-------- Exemple des Membres static de la classe ShootsMERGE --------//

// Declaration //
protected: static char tabParamResult[][20];
public: static boolean_t containsParamResult(const char *);

// Definition //
/** Array of the attributes of ShootsMERGE */
char ShootsMERGE::tabParamResult[][20] =
{"maxOrder","maxSideBranches0","lgi0","noLgiRankdep","lgi0_rank","lgi_etio",
 "lgi_rank_dep","lgi_order_dep","noLgRankdep","l0_rank","l_etio","l_rank_dep",
 "l_order_dep","lg_branch_max","Ltbs","baseAngle","topAngle","internode_radius0",
 "internode_density","internode_c","brpattern","branching_phy_age","frac_int",
 "resHeight","resIntLength","resIntMass","resLeafMass","resIntRadius","resBrNb" };

/** Controle that a parameter or a result is in the table (belong to the model)*/
boolean_t ShootsMERGE::containsParamResult(const char * inStr)
{
   for(int i=0;i<sizeof(tabParamResult)/20.0;i++)
      if (strcmp(tabParamResult[i],inStr)==0)
         return true;
   return false;
}
```

Figure 18 : Extrait de code des membres static de ShootsMERGE

Une fois ce système en place, il ne reste plus qu'à insérer ces contrôles dans la fonction de chargement du fichier. Encore une fois un problème se pose : comment appeler les fonctions de recherche *static* depuis la méthode de chargement alors qu'UNIF est une plateforme générique qui ne doit donc pas avoir de lien avec Gemini. En agissant sur UNIF directement il est possible de contourner le problème sans appeler directement les fonctions de recherche, un peu à l'image du travail de Luc Touraille [TOURAILLE 2007]. On définit donc dans la classe *Node* une méthode virtuelle pure: `boolean_t inspectClass(const char * name)` qui sera redéfinie dans les sous-modules. La classe *Node* étant l'entité « de base », déclarer cette méthode à l'intérieur permet de toucher tous les modules de GEMINI grâce à l'héritage. En effet pour un module implémenté différemment par les deux versions, on cherchera dans la version « non active » si le paramètre qui pose problème s'y trouve. Si c'est le cas alors celui-ci sera ignoré, sinon c'est qu'il représente en fait une erreur. Par exemple dans *eom_compartment* de Soilopt, la méthode retournera faux tout simplement en effet ce module est le même pour les deux versions. Pour un module à 2 versions tels que *ShootsINRA*, la méthode de recherche *static* de *ShootsMERGE* sera lancée pour contrôler que le paramètre lui appartient. Pour illustrer cette explication voici la fonction *inspectClass()* implémentée par la classe *ShootsMERGE* :

```
//-----------------------------------------InspectClass

/** Search for the parameter or the result in the model */
boolean_t ShootsMERGE::inspectClass(const char * _name)
{
    return ( (ShootsINRA::containsParamResult(_name))  ||
             (ConcreteShoots::containsParamResult(_name)) ||
             (ConcreteLeaf::containsParamResult(_name)));
}
```

Figure 19 : Code de la fonction inspectClass() implémentée dans ShootsMERGE

Cette méthode sera appelée pour chaque paramètre qui n'est pas actif dans le modèle. Grâce à cette recherche, lors du chargement le programme pourra faire la différence entre un paramètre qui est présent dans le modèle mais qu'il ne doit pas utiliser et une simple erreur qu'il doit signaler.

Le système de chargement des fichiers et les fichiers eux-mêmes ont donc été mis à jour et permettent d'utiliser indifféremment les deux modélisations avec le même fichier de paramétrage en changeant simplement la valeur du paramètre de choix du modèle. Une fois tout ceci effectué, le programme m'a permis de générer très rapidement un grand nombre de fichiers (133) puis de les charger indifféremment de la version pour tester correctement le modèle.

1.2. Evolution vers un outil plus récent

Suite à ces travaux sur le modèle, nous avons décidé d'effectuer un portage vers une version plus récente de Borland, et ce pour plusieurs raisons :

- L'équipe avait besoin d'acheter de nouvelles licences et il était dommage d'investir dans un logiciel maintenant dépassé.
- L'interface dans C++ Builder 6 est peu ergonomique et surtout ne dispose pas des outils tels que la coloration syntaxique ou la signalétique des parenthèses et accolades.
- Le compilateur de C++ Builder 6 présente des insuffisances comparé aux compilateurs plus récents (paramètre non utilisés, dépassement de tableau…)
- Enfin, C++ Builder 6 est plutôt instable et présente de sévères lacunes dans certains de ses outils, comme par exemple l'absence de bouton pour stopper une session de débogage.

Nous avons donc décidé de porter GEMINI sous la dernière version de C++ Builder, sortie en Avril dernier. Outre l'interface beaucoup plus intuitive, pratique et ergonomique, de nombreuses fonctionnalités ont été ajoutées comme par exemple la mise à jour des fonctions de la VCL, un débogueur amélioré et la prise en charge du format Unicode qui vient donc « concurrencer » le format ASCII. C'est d'ailleurs cette dernière qui, même si elle représente une grande force pour le logiciel (principalement pour la programmation sous Windows), nous a posé de gros soucis techniques.

GEMINI, et plus particulièrement UNIF dans le cas présent, ont été implémentés il y a de cela six ans. De nouveaux types de données ont fait leur apparition (wchar_t…) et de nombreuses fonctions présentes à l'époque dans les bibliothèques de la STL ont été remplacées ou modifiées : leur signature a changé, le type de retour aussi… Le portage était donc loin d'être trivial. Il fallait réussir à trouver les bonnes fonctions pour pouvoir compiler sous Borland sans pour autant rendre impossible la compilation sous Linux en utilisant des composants Borland. Pour cela, nous avons essayé de séparer au maximum les utilisations sous les deux environnements, en créant des macros différentes pour le chargement des fichiers (paramétrage et annexes). En effet, les fonctions de manipulation des chemins et des dossiers font partie des méthodes qui ont été modifiées sous Borland, utilisant des chaînes de caractères au format Unicode (wchar_t * permettant de gérer des accents). Il fallait donc dissocier les macros de chargement des fichiers : l'une est utilisée pour Linux et l'autre pour Windows. Nous les avons ensuite implémentées pour chacun des deux systèmes, à l'aide d'un bloc *if...else* sur la nature du compilateur utilisé (g++ ou Borland).

Cette modification des macros de chargement des fichiers permet au programme d'être à nouveau opérationnel avec les deux environnements et nous permet donc d'utiliser GEMINI dans un environnement bien plus agréable et surtout plus stable pour les développements à venir.

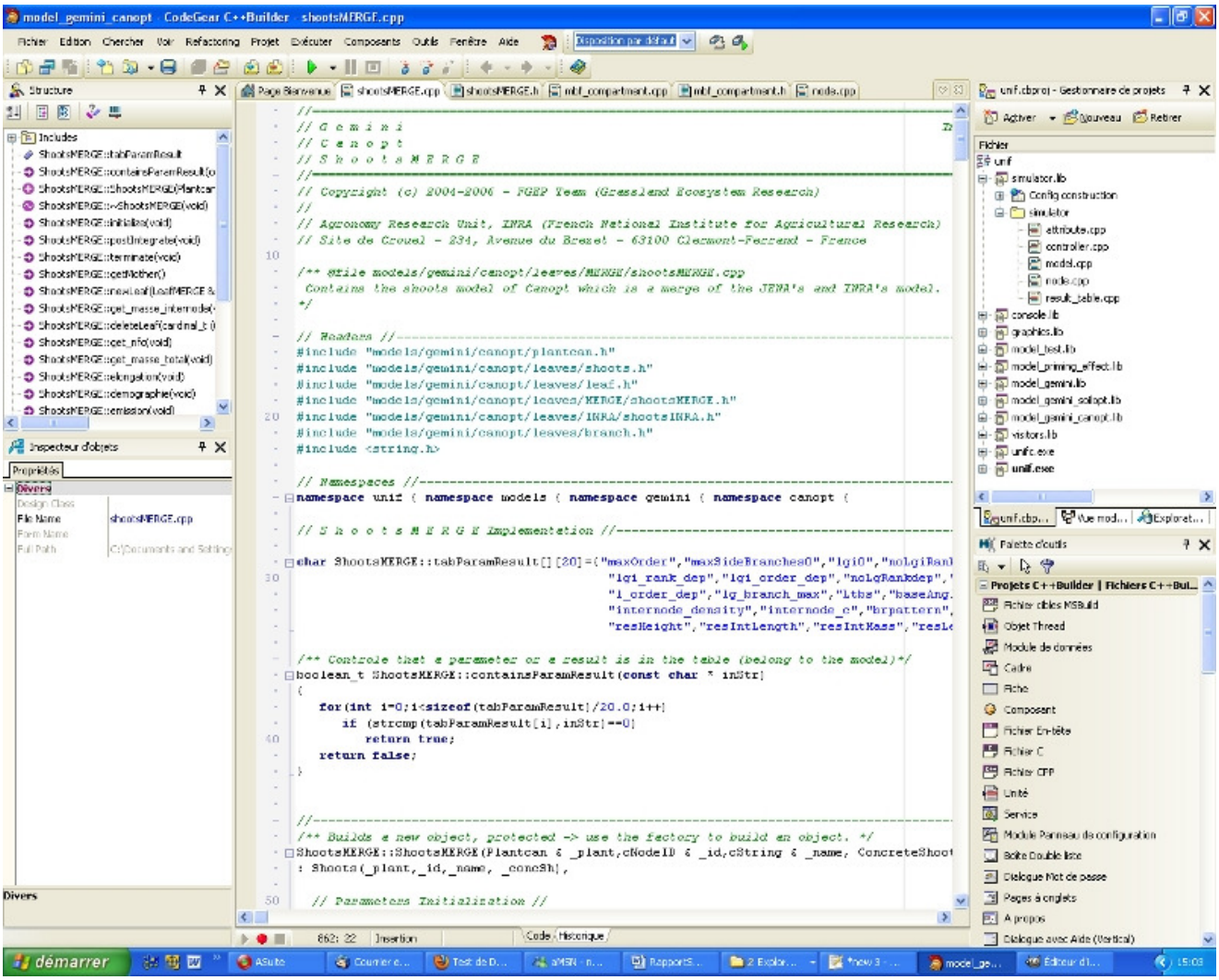


Figure 20 : Interface de C++ Builder 2009

Avec cette toute nouvelle version et ce nouveau compilateur, nous nous sommes donc lancés dans le développement de la dernière partie de mon stage : l'ajout des flux d'eau dans le modèle GEMINI.

2. Intégration de l'eau

L'ajout de la modélisation des flux d'eau au travers du sol et de la plante dans GEMINI est une partie importante de mon stage, elle m'a permis de suivre une démarche de recherche scientifique dans son intégralité sur un mécanisme écophysiologique. En effet, peu de choses avaient été réalisées sur l'eau dans le modèle et il a fallu commencer par étudier et comprendre les théories existantes sur ce phénomène, notamment les mécanismes régissant les teneurs en eau du sol et de la plante. Cela a nécessité une recherche d'articles sur les travaux déjà effectués. Ensuite, la seconde étape a porté sur les différentes possibilités de modélisation et enfin sur la procédure d'implémentation.

2.1. Mécanismes de l'évapotranspiration

La teneur en eau du sol et le volume d'eau absorbé par la plante détermineront en partie sa croissance et son maintien. En cas de forte carence en eau du sol consécutif à une faible teneur en eau, la plante évolue vers un état de stress hydrique, modifiant fortement la physiologie de celle-ci et s'il persiste trop longtemps pouvant entraîner sa mort. Il est alors important de bien décrire l'évolution de la teneur en eau du sol si l'on veut comprendre son impact sur les processus végétaux. Cette évolution est extrêmement liée au mécanisme d'évapotranspiration. Il résulte de la combinaison de deux phénomènes distincts, d'une part l'évaporation de l'eau au niveau du sol, et d'autre part la transpiration de l'eau au niveau de la plante :

- L'évaporation est le phénomène par lequel de l'eau liquide est transformée en vapeur dans les interstices du sol et rejoint l'eau de l'atmosphère
- La transpiration est la vaporisation de l'eau liquide contenue dans les tissus de la plante par les stomates.

Ces deux processus ont lieu simultanément, il est donc très difficile de les caractériser formellement. On sait néanmoins grâce à des études poussées que les conditions environnementales sont très importantes (température, rayonnement, vent, teneur en eau du sol, humidité relative de l'air, …). Par exemple, l'évaporation et la vaporisation dépendent du rayonnement solaire à la surface du sol et des feuilles. Ainsi, quand la végétation est dense elle subira la majeure partie du rayonnement et donc la transpiration sera plus importante. Si au contraire la végétation est peu dense alors le sol subit un plus fort rayonnement et donc l'évaporation sera plus importante.

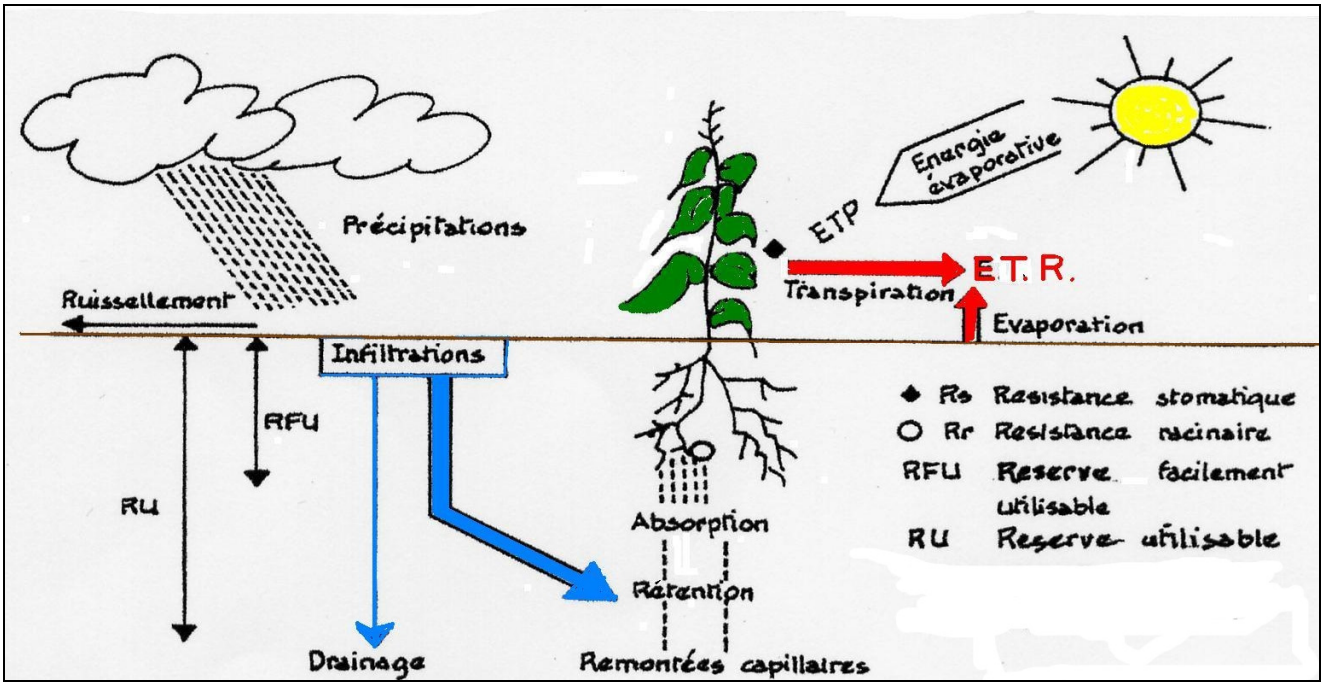


Figure 21 : Cycle de l'eau dans le sol et la plante

Avec l'appui de Jean-François Soussana et Vincent Maire, l'étude de nombreux articles ([ALLEN et al. 1998], [TUZET et al. 2003], [SINCLAIR 2005], [DEWAR 2002]) nous a permis de trouver deux modèles d'évapotranspiration avec des visions différentes de la représentation du mécanisme.

Le modèle de Sinclair ([SINCLAIR 2005]) associé au modèle de Penman-Monteith ([ALLEN et al. 1998]) donne une vision macroscopique du phénomène : le modèle calcule l'évapotranspiration de toute la canopée sans tenir compte des caractéristiques de chacune des espèces végétales. A partir de fonctions de pédotransfert, relatives à la composition du sol (taux en argile, limon et sable), il est possible de connaître le comportement moyen de l'eau dans le sol. Puis, en connaissant l'environnement climatique moyen et la couverture de la végétation, il est possible de calculer une évapotranspiration potentielle puis effective en fonction de la teneur en eau du sol et de la plante. A titre d'information, ce modèle est utilisé et recommandé par la FAO (Food and Agriculture Organization) pour connaître le besoin des cultures pures (Blé, Maïs, Riz, …) en eau.

Le modèle de Tuzet discrétise la canopée en couches et tient compte des caractéristiques de chacune des espèces composant la canopée. Il est plus précis que le modèle de Sinclair puisqu'il ne calcule pas l'évapotranspiration globale directement, mais la transpiration de chaque couche de la canopée pour ensuite les sommer à l'échelle du couvert entier. Ce modèle tient compte de l'ouverture stomatique des feuilles contrairement au précédent. L'ouverture stomatique rend compte de la diffusivité de l'eau depuis la plante jusqu'à l'atmosphère. La vitesse d'ouverture des stomates et le seuil de teneur en eau du sol au delà duquel la plante est en déficit hydrique sont des spécificités de chaque espèce végétale. Néanmoins si le modèle est plus précis, il est aussi plus dur à mettre en œuvre, avec un nombre de paramètres plus important.

Nous avons donc décidé d'utiliser les capacités de chacune de ces approches pour calculer l'évapotranspiration de la communauté (ANNEXE 1 et 2) et résoudre une difficulté de ces deux modèles hydrique, le calcul du potentiel matriciel de l'eau dans une feuille : Ψ_{leaf}. Cette variable d'état, si l'on veut correctement la modéliser implique de connaître un nombre important de paramètres pour chacune des espèces. Comme chacun des deux modèles dépend de cette variable, une méthode d'optimisation permettait de résoudre cette difficulté sans avoir besoin de nouveaux paramètres. Après avoir réfléchi aux différentes possibilités qui s'offraient à nous, nous avons décidé d'utiliser une méthode de convergence des modèles en faisant varier la valeur de cette constante dans un intervalle et avec un pas donné. A partir des résultats de chacun des modèles, il est possible de

calculer Ψ_{leaf} en minimisant la différence absolue entre les deux évapotranspirations potentielles. Application des modèles et méthode de convergence

Comment réaliser l'étude de la (possible) convergence de ces modèles complexes ? Il nous fallait un outil simple afin d'implémenter les modèles rapidement et en même temps puissant car il devait regrouper toutes les fonctionnalités dont nous avions besoin. Le logiciel Berkeley Madonna s'est imposé comme étant le plus adéquat. Berkeley Madonna est un solveur d'équation différentielle et un outil d'optimisation mathématique écrit en C par l'université de Berkeley. Il est utilisé par les institutions académiques et les laboratoires de recherche pour la construction de modèles mathématiques. Il contient toute une panoplie d'outils mathématiques et de fonctionnalités d'optimisation. Il intègre de plus une interface graphique écrite en JAVA (Flowchart) qui permet de l'utiliser un peu à la manière d'un réseau de neurones. Chaque paramètre, chaque opération sont identifiés dans une « bulle » qui peut posséder des flux d'entrée/sortie. Toutes les variables utilisées pour effectuer le calcul dans la bulle doivent être connectées, tandis que les flux de sortie contiendront le résultat du calcul. Ainsi, n'importe quel système complexe peut devenir très simple, rapide à implémenter tout en conservant une grande vitesse d'exécution (ANNEXE 4).

Une fois les deux modèles implémentés avec Berkeley, nous avons calculé la valeur de l'évapotranspiration de chaque modèle pour des valeurs de Ψ_{leaf} comprises dans l'ntervalle [-5 ; 0] non compris par pas de 0.1. Un graphe en sortie (figure 22) affiche les résultats de chacun des modèles et la valeur absolue de la différence en fonction de la valeur de Ψ_{leaf}. A partir de cette courbe nous avons réussi à trouver un point « de convergence » où les résultats semblaient assez proches. La valeur correspondante de Ψ_{leaf} est donc devenue notre valeur de référence (Ψ_{leaf} = -1.6 sur le graphique ci-dessous), à utiliser dans le modèle GEMINI. Une phase de test s'en est suivie pour contrôler la réactivité et la réponse des deux modèles à des variations de valeur de différents paramètres nous permettant de déterminer s'il était possible de les implémenter directement dans GEMINI.

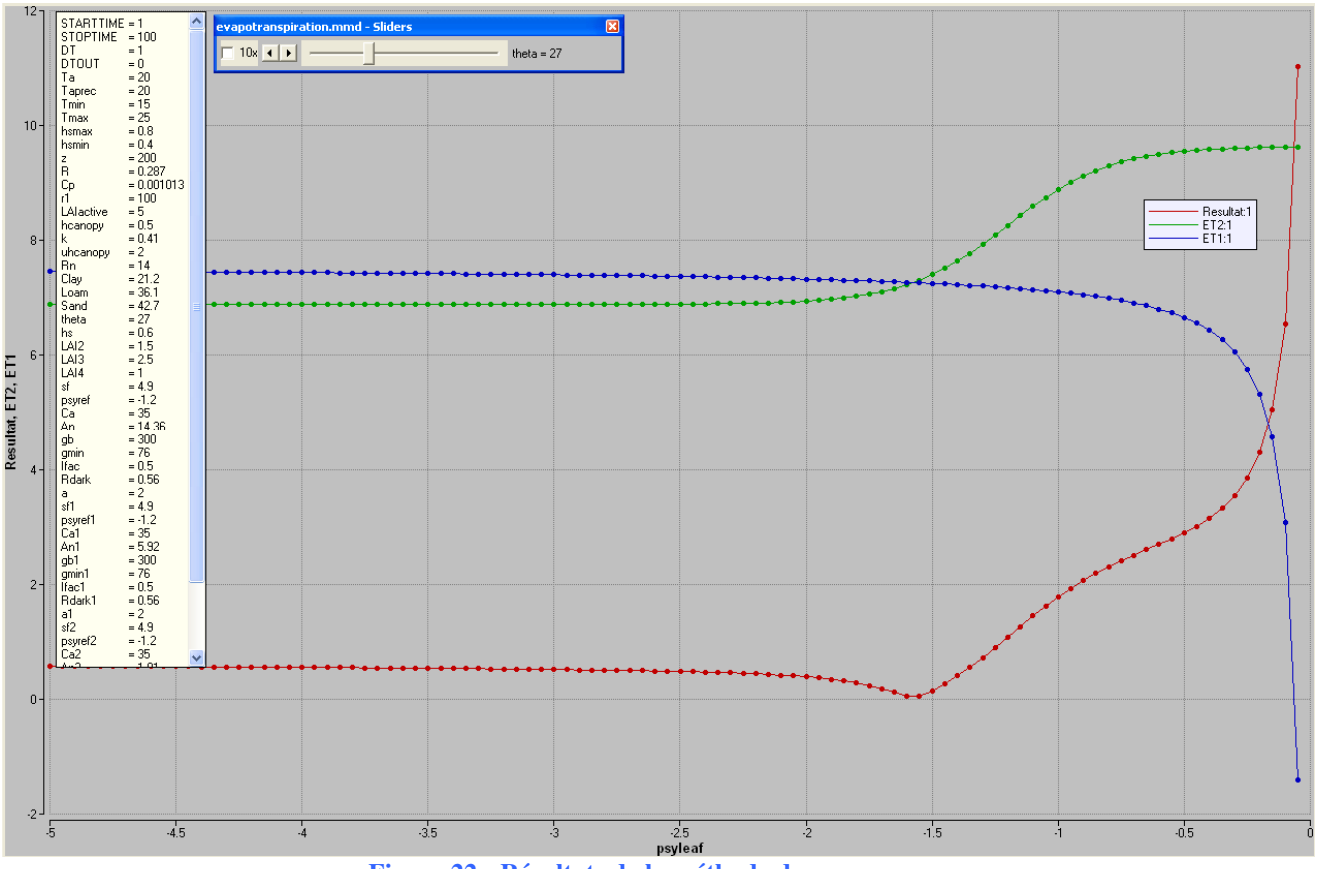


Figure 22 : Résultats de la méthode de convergence

Grâce à ce travail, nous avons pu trouver une convergence pour une certaine valeur de Ψ_{leaf}. Pour pouvoir coupler ces deux modèles et donc calculer au mieux l'évapotranspiration dans GEMINI, il nous faut à chaque itération, trouver le meilleur Ψ_{leaf} possible c'est-à-dire celui pour lequel les modèles fournissent des résultats concordants. J'ai donc étudié différents algorithmes possibles pour réaliser cette optimisation à chaque itération, plusieurs possibilités sont apparues, efficace chacune suivant un contexte particulier.

- Une boucle « for » sur Ψ_{leaf} permet de calculer plusieurs fois le résultat des modèles pour étudier la valeur absolue de leur différence. Cette méthode est bien sur simplissime, et ne nécessite de connaitre que l'intervalle de valeurs de Ψ_{leaf} (minimum, maximum et pas). Néanmoins celle-ci est peu efficace et s'alourdie avec le nombre de valeurs possibles de Ψ_{leaf}.
- Une méthode de dichotomie donnerait ici d'excellents résultats. En recherchant le point où la courbe Résultats (en rouge sur la figure 22) s'annule. La complexité de cette méthode est bien meilleure que la précédente : log(N) au lieu de N.
- Un algorithme de type recuit simulé. Le recuit simulé est une métaheuristique inspirée d'un processus utilisé en métallurgie qui permet (en optimisation) de trouver les extrema d'une fonction. Assez difficile à mettre en œuvre ce type d'algorithme est extrêmement performant pour des problèmes complexes.
- Un algorithme de recherche mathématique telle que la méthode du gradient conjugué.

Au regard du contexte et des objectifs de notre démarche, deux possibilités sont à éliminer directement. En effet comme on peut le voir sur la courbe de la figure 22, la fonction des résultats n'est pas monotone, elle peut donc comporter des extrema locaux qui empêchent le déroulement de la méthode dichotomique. Une métaheuristique basée sur une descente stochastique pourrait être utilisée pour éviter le piège des extrema locaux mais la taille de l'intervalle de valeurs de Ψ_{leaf} ne justifie pas son utilisation. De plus le modèle GEMINI ne serait alors plus un modèle déterministe. Reste alors la possibilité de la boucle « for » bien que peu élégante et optimisée, et un algorithme mathématique déterministe d'étude de fonctions. Je me suis donc bien évidemment lancé dans la recherche d'une méthode de ce type susceptible de nous aider.

Après de longues recherches et l'étude comparative de plusieurs méthodes, j'ai implémenté les deux modèles et la méthode de Brent dans un mini-projet avec des valeurs fixes afin de pouvoir tester les résultats sans intervenir sur GEMINI. La méthode de Brent est une méthode de recherche d'un zéro d'une fonction (convertie ici en minimisation) combinant la méthode de dichotomie, la méthode de la sécante et l'interpolation quadratique inverse. J'ai utilisé la fonction *clock()* permettant de d'avoir le temps CPU à un instant donné. En soustrayant deux valeurs de *clock()* au début et à la fin de la fonction on peut donc avoir le temps CPU que prend l'exécution de la méthode. Les résultats sont donnés dans le tableau ci-dessous suivant les valeurs de thêta, un paramètre des modèles :

Thêta	18	27	40
Boucle « For »			
Temps CPU	176	187	188
Résultat	-8.66269	-3.22616	-3.10188
Méthode de minimisation de Brent			
Temps CPU	0	0	0
Résultat	-8.66242	-3.22634	-3.10698

Figure 23 : Comparaison des méthodes en temps CPU

Le tableau de la figure 23 démontre donc clairement l'efficacité de la méthode de Brent et fait d'elle la priorité pour l'implémentation. Néanmoins nous allons, certainement dans les prochains jours, réaliser une analyse de sensibilité du modèle pour déterminer les paramètres qui ont un effet important sur les résultats. Après quoi nous lancerons de nombreux tests sur le petit modèle que j'ai réalisé en faisant varier les paramètres importants révélés par l'analyse afin de vérifier qu'aucune erreur n'apparait.

Quelques détails restent cependant à préciser comme par exemple le « lieu » d'intégration du mécanisme dans GEMINI. Deux possibilités se présentent : la première est d'implémenter les équations des modèles dans les méthodes de pré-intégration respectivement du sol et de la plante et séparer l'algorithme d'optimisation. La seconde est de considérer l'eau comme une entité et donc d'implémenter tous les mécanismes dans un nouveau module. La deuxième solution est à mon sens la meilleure. En effet elle laisse une plus grande marge de manœuvre et permettra d'utiliser ou non en ajoutant le module et évitera l'éparpillement de l'algorithme d'optimisation et du modèle qu'il utilise. Néanmoins la question reste pour l'instant à trancher.

Perspectives et discussion

Le modèle GEMINI et la plateforme UNIF continuent leur évolution en faisant l'objet presque chaque année d'un stage d'un étudiant de l'ISIMA. Bien qu'écrit à la base pour GEMINI, on peut espérer que de nouveaux modèles seront bientôt implémentés sur UNIF pour tirer parti de sa généricité. GEMINI devient lui de plus en précis et réaliste en peaufinant la modélisation par l'intégration de nouveaux mécanismes comme les flux d'eau dans le sol et la plante.

Pour l'heure l'implémentation de ce phénomène n'a pas encore été complètement terminée et cette fonctionnalité n'est donc pas opérationnelle. C'est durant le mois de Septembre, le dernier de mon stage, que je m'évertuerai à la terminer pour clore la longue démarche scientifique qui fut nécessaire. A court terme, l'équipe pense implémenter la compétition des plantes pour l'eau qui n'est pour l'instant considérée que pour chaque individu séparément.

En ce qui concerne la refonte du programme nécessaire à la prise en compte de la modélisation de type « MERGE », celle-ci est pleinement opérationnelle. Néanmoins le développement annexe apporté à la plateforme UNIF pour pouvoir charger les paramètres des deux modèles sans erreur n'est peut-être pas la meilleure solution. C'est néanmoins la seule que nous avons pu trouver et mettre en place dans l'immédiat mais la piste d'une implémentation en *template* de l'introspection mériterait d'être creusée. La modélisation « INRA » est parfaitement stable grâce à nos corrections de nombreuses erreurs, mais la modélisation « MERGE », même si elle est utilisable n'est pas encore validée puisque le passage à la version 2009 de C++ Builder a permis de déceler plusieurs erreurs importantes dans le code source. Ces erreurs sont en cours de correction par l'équipe allemande du Max Planck Institute.

A plus long terme le modèle GEMINI pourra certainement se développer encore plus en agrandissant l'échelle de la simulation en passant par exemple à une modélisation en 3 dimensions des interactions plante-plante et plante-sol et en ne se limitant plus à un patch de végétation de dimension restreinte.

Conclusion

J'ai effectué mon stage de deuxième année au sein de l'Unité de Recherche sur l'Ecosystème Prairial de l'INRA de Crouël. Ce stage avait plusieurs objectifs : dans un premier temps réaliser une version hybride du modèle GEMINI regroupant les différentes versions précédemment développées en mutualisant au maximum les données communes pour faciliter les développements futurs ; puis dans un deuxième temps étudier un mécanisme biologique et l'intégrer au modèle.

Au terme de ce stage le premier objectif a été pleinement rempli. Ceci permettra de faciliter le test et la validation de la modélisation de type « MERGE » une fois les problèmes soulevés réglés puis par la suite les développements qui impacteront sur les deux versions. La modélisation de l'eau, bien qu'encore inachevée est très avancée et sera surement terminée à la fin du mois de Septembre, date de mon départ. J'espère que ce degré de complexité supplémentaire au programme donnera des résultats encore plus proches de la réalité lors des tests finaux.

Pour faciliter les développements à venir, j'ai ajouté au manuel de programmation qui accompagne UNIF et GEMINI, que tous les stagiaires depuis le début de ce projet ont complété, le mode de gestion des paramètres actifs et inactifs dans le chargement de fichiers. De plus comme depuis le début, le code source a été en permanence commenté afin de garder la documentation Doxygen la plus complète possible, ce qui est certainement un des points forts de GEMINI.

Sur le plan personnel, ce stage a été une expérience très enrichissante pour plusieurs raisons. Tout d'abord j'ai pu améliorer ma connaissance du C++, la pratique quotidienne durant le stage m'a permis d'approfondir les bases que j'avais déjà mais aussi d'aborder plusieurs aspects que je n'avais jamais vus notamment de nombreuses classes et structures permettant de faciliter considérablement l'implémentation. De plus le travail en équipe avec des biologistes non informaticiens est une expérience très enrichissante puisqu'elle m'a permis de rester à leur écoute pour comprendre leurs attentes, donner mon point de vue et mon avis sur la faisabilité de tel ou tel point. De la même manière ils m'ont beaucoup expliqué les mécanismes aussi bien mathématiques que biologiques pour que je les comprenne et retranscrive au mieux, les échanges et la communication au sein de l'équipe étaient donc primordiaux pour avancer dans le travail en suivant nos objectifs. Cela m'a permis d'acquérir une certaine méthodologie de travail due à la démarche scientifique à tenir. Enfin, le fait d'être dans un institut m'a permis de mieux appréhender le monde de la recherche et est une très bonne expérience que je pourrai confronter avec celle de mon stage en entreprise en 3ème année et qui m'aidera à déterminer mon orientation future.

Références bibliographiques

[ALLEN et al. 1998]
Richard G. ALLEN, Luis S. PEREIRA, Dirk RAES
"Crop Evapotranspiration – Guidelines for computing crop water"
FAO – Food and agriculture Organization of the United Nations. Irrigation and drainage paper 56
1998

[COQUILLARD et HILL 1997]
Patrick COQUILLARD et David R.C. HILL
"Modélisation et simulation d'écosystèmes – Des modèles déterministes aux simulations à événements discrets"
Paris : MASSON, 1997

[DEWAR 2002]
R.C. DEWAR
"The Ball-Berry-Leuning and Tardieu-Davies stomatal models: synthesis and extension within a spatially aggregated picture of guard cell function"
Plant, Cell and Environment n°25
2002

[GRIMBICHLER 2005]
David GRIMBICHLER, "Réimplémentation et évolution du modèle GEMINI sur la plateforme UNIF"
Rapport de stage, 2005

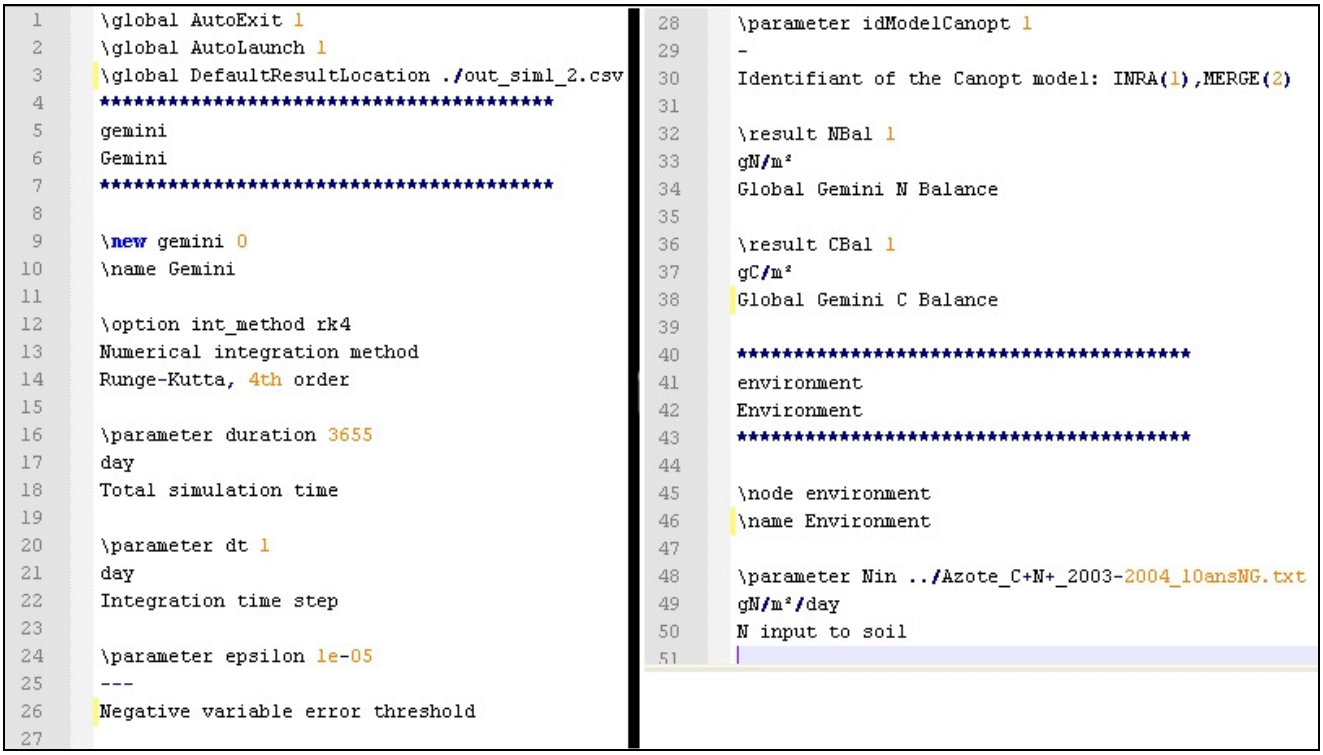
[SINCLAIR 2005]
Thomas R. SINCLAIR
"Theoretical Analysis of Soil and Plant Traits Influencing Daily Plant Water Flux on Drying Soils"
Agronomy Journal n° 97
2005

[TOURAILLE 2007]
Luc TOURAILLE, "Simulation de la croissance de plantes en communauté dans un écosystème prairial"
Rapport de stage, 2007

[TUZET et al. 2003]
A. TUZET, A. PERRIER, R. LEUNING
"A coupled model of stomatal conductance, photosynthesis and transpiration"
Environnement et Grandes Cultures – INRA
2003

[WITZMANN 2004]
Stéphane WITZMANN, "Réimplémentation du modèle GEMINI sur la plateforme UNIF"
Rapport de stage 2004

ANNEXE 1 : Les fichiers de paramétrage de GEMINI



```
1  \global AutoExit 1
2  \global AutoLaunch 1
3  \global DefaultResultLocation ./out_siml_2.csv
4  ****************************************
5  gemini
6  Gemini
7  ****************************************
8
9  \new gemini 0
10 \name Gemini
11
12 \option int_method rk4
13 Numerical integration method
14 Runge-Kutta, 4th order
15
16 \parameter duration 3655
17 day
18 Total simulation time
19
20 \parameter dt 1
21 day
22 Integration time step
23
24 \parameter epsilon 1e-05
25 ---
26 Negative variable error threshold
27
```

```
28 \parameter idModelCanopt 1
29 -
30 Identifiant of the Canopt model: INRA(1),MERGE(2)
31
32 \result NBal 1
33 gN/m²
34 Global Gemini N Balance
35
36 \result CBal 1
37 gC/m²
38 Global Gemini C Balance
39
40 ****************************************
41 environment
42 Environment
43 ****************************************
44
45 \node environment
46 \name Environment
47
48 \parameter Nin ../Azote_C+N+_2003-2004_10ansNG.txt
49 gN/m²/day
50 N input to soil
51
```

On voit, sur la figure, que le fichier commence tout d'abord par une suite de variables « globales », c'est-à-dire de valeurs nécessaires à UNIF et non spécifique à une espèce, telles que l'arrêt et le lancement automatique de la simulation, le nom du fichier de sortie (pour les résultats) …

Tous les modules sont ensuite détaillés et précédés, par convention et ce afin d'accroître la clarté du fichier, par un bloc d'étoiles contenant leur nom et leur classe. Toutefois, au chargement, UNIF ne prend en compte que les lignes commençant par un '\'. Un module qui possède des sous-modules (comme gemini) est précédé de new et suivi d'un numéro. Ce numéro est son indice (ou rang) dans l'arborescence des modules par rapport au module créé juste avant lui, ce qui permettra dans la fonction de chargement de retrouver le « père » d'un sous-module. En effet un sous-module n'est pas forcément à la suite de son « père » dans le fichier il faut donc pouvoir remonter jusqu'à lui. « \node » quant à lui précède un module qui n'a pas de sous-modules, comme environnement par exemple. Au sein de chaque bloc module se trouvent les attributs \parameter, \option et \result identifiés de cette façon et suivi de la valeur ou du nom du fichier qui contient les valeurs (ex : ligne 48). Le paramètre que j'ai rajouté pour choisir quelle version du modèle utilisée est à la ligne 28. Il appartient au module Gemini. Celui étant le premier dans le fichier (le plus haut) il est créé en premier et ses attributs sont chargés juste après. Lorsque Canopt sera instancié, le paramètre sera déjà initialisé et le module pourra donc créer la Factory correspondante (INRA ou MERGE) qui s'occupera alors de créer les Shoots (module suivant Canopt).

ANNEXE 2 : Modèle de Sinclair

Table 1: Table des équations du modèle de Sinclair pour le calcul de l'évapotranspiration de toute la canopée

Process	Equation	unit
Actual evapotranspiration	$ET_l = RT \cdot ET_0$	mm day^{-1}
Pedotransfert functions	$b = 0.157 \cdot Clay - 0.003 \cdot Sand + 3.1$	dimensionless
	$\theta_{sat} = 0.037 \cdot Clay - 0.142 \cdot Sand + 50.5$	cm^3 cm^{-3}
	$\Psi^*_{soil} = 10^{0.54 - 0.0095 \cdot Sand + 0.0063 \cdot Loam}$	MPa
Soil water potential	$\Psi_{soil} = \Psi^*_{soil} \cdot \left(\theta / \theta_{sat} \right)^{-b}$	mmol m^{-2} s^{-1}
Soil hydraulic conductivity	$K = K_{sat} \cdot \left(\theta / \theta_{sat} \right)^{2b+3}$	cm d^{-1}
Relative transpiration	$RT = 1 - \Psi^*_{soil} \cdot \left(\theta / \theta_{sat} \right)^{-b} / \Psi_{leaf}$	dimensionless
Potential evapotranspiration	$ET_0 = \dfrac{\Delta \cdot (R_n - G) + \rho_a \cdot C_p \cdot (e_s - e_a) / r_a}{\Delta + \gamma \cdot (1 + r_s / r_a)}$	mm day^{-1}
Mean air density at constant pressure	$\rho_a = P / T_K \cdot R$	kg m^{-3}
Air specific heat	$\gamma = C_p \cdot P / \varepsilon \cdot \lambda = 0.65 \cdot P$	MJ kg^{-1} °C^{-1}
Partial pressure	$P = 101.3 \cdot \left(293 - 0.0065 \cdot z / 293 \right)^{5.26}$	kPa
Aerodynamic resistance	$r_a = \ln\left(\dfrac{z_m - d}{z_{om}} \right) \cdot \ln\left(\dfrac{z_h - d}{z_{oh}} \right) \cdot \dfrac{1}{k^2 \cdot u_{h_{canopy}}}$	s m^{-1}
	$z_m = z_h = h_{canopy}$	m
	$d = 2/3 \cdot h_{canopy}$; $z_{om} = 0.123 \cdot h_{canopy}$; $z_{oh} = 0.1 \cdot z_{om}$	
Bulk surface resistance	$r_s = r_l / LAI_{active}$	s m^{-1}
Slope of vapour pressure curve	$\Delta = 4099 \cdot e_a / (T_a + 237.3)^2$	kPa °C^{-1}
Actual vapour pressure	$e_s = \left[e^0(T_{min}) \cdot h_{s_{max}} + e^0(T_{max}) \cdot h_{s_{min}} \right] / 2$	kPa
Saturation vapour pressure deficit	$e^0(T) = 0.611 \cdot \exp(17.27 \cdot T_a / T_K)$	kPa
	$e_s = \left[e^0(T_{max}) + e^0(T_{min}) \right] / 2$	
Soil heat flux	$G = 0.38 \cdot \left(T_{a_t} - T_{a_{t-1}} \right)$	MJ m^{-2} day^{-1}

Table 2 : Variables et paramètres utilisés dans les équations de la Table 1.

Symbol	unit	value	Description
Variable			
RT	0-1		Relative transpiration
ET_0	mm day^{-1}		Potential evapotranspiration
R_n	MJ m^{-2} day^{-1}		net radiation at the crop surface
G	MJ m^{-2} day^{-1}		soil heat flux density
e_a	kPa		actual vapour pressure
e_s	kPa		saturation vapour pressure
e_s - e_a	kPa		saturation vapour pressure deficit
Δ	kPa $°C^{-1}$		slope vapour pressure curve
γ	kPa $°C^{-1}$		psychrometric constant
r_a	s m^{-1}		aerodynamic resistance
ρ_a	kg m^{-3}		mean air density at constant pressure
r_a	s m^{-1}	$208/u_2$	aerodynamic resistance
z_m	m		height of wind measurements
z_h	m		height of humidity measurements
d	m		zero plane displacement height
h_{canopy}	m		Height of the canopy
z_{om}	m		roughness length governing momentum transfer
z_{oh}	m		roughness length governing transfer of heat and vapour
θ_{sat}	cm^3 cm^{-3}	0.62	Volumetric water content m3 m_3 at saturation in soil
Ψ^*_{soil}	MPa	97.10^{-3}	Soil water potential at saturation
b	dimensionless	9	Empirical constant for each soil
r_s	s m^{-1}	70	(bulk) surface resistance
LAI_{active}	m^2 m^{-2}		active (sunlit) leaf area index
$e°(T_{min})$	kPa		saturation vapour pressure at daily minimum temperature
$e°(T_{max})$	kPa		saturation vapour pressure at daily maximum temperature
P	kPa		atmospheric pressure
Ψ_{leaf}	MPa		Leaf water potential at saturation
Input Variable			
T_k	°K		mean daily air temperature
T_a	°C		mean daily air temperature
u_z	M s^{-1}		wind speed at height z
hs_{max}	0-1		maximum relative humidity
hs_{min}	0-1		minimum relative humidity
z	m		Elevation above sea level
θ	cm^3 cm^{-3}		Volumetric water content
Parameter-Constant			
ε	dimensionless	0.622	ratio molecular weight of water vapour/dry air
r_l	s m^{-1}	100	bulk stomatal resistance of the well-illuminated-watered leaf
λ	MJ kg^{-1}	2.45	latent heat of vaporization
k	dimensionless	0.41	von Karman's constant
C_p	MJ kg^{-1} $°C^{-1}$	$1.013\ 10^{-3}$	specific heat at constant pressure
R	kJ kg^{-1} K^{-1}	0.287	specific gas constant

ANNEXE 3 : Modèle de Tuzet

Table 1 : Equations du modèle de Tuzet, vision écophysiologique, calcul par couche

Process	Equation	Unit
By layer		
Stomatal conductance	$g_s = g_{min} + g_{fac} \cdot (A_n + I_{fac}.R_{dark}) \cdot 10^2 \cdot \frac{h_s}{C_s}$	mmol m^{-2} s^{-1}
CO_2 partial pressure at the leaf boundary layer	$C_s = C_a - A_n \cdot 10^2 / g_b$	Pa
Stomatal sensitivity	$G_{fac} = \frac{1+\exp[sf \cdot \Psi_{ref}]}{1+\exp[sf \cdot (\Psi_{leaf} - \Psi_{ref})]}$	dimensionless
Transpiration	$E = 1.6 \cdot g_s \cdot (e_s - e_a)$	mmol m^{-2} leaf s^{-1}
For whole canopy		
Evapotranspiration	$ET_2 = \sum_z^m E_z \cdot LAI_z$	mmol m^{-2} soil s^{-1}

Table 2 : Variables et paramètres utilisés dans les équations de la Table 1.

Symbol		Unit	Description
Parameter			
g_b	mmol m^{-2} s^{-1}	300	Leaf boundary layer conductance to water vapour
g_{min}	mmol m^{-2} s^{-1}	76	Minimal conductance to water vapour
Ψ_{ref}	MPa	-1.2	Reference leaf water potential (value for high sensitive plant to water stress)
sf	dimensionless	4.9	Sensitivity parameter
Variable			
g_s	mmol m^{-2} s^{-1}		Stomatal conductance to water vapour
I_{fac}	dimensionless		Coefficient representing the extent to which R_{dark} is inhibited in the light
A_n	µmolC m^{-2} leaf s^{-1}		Net photosynthesis per unit leaf area (import Gemini layer data)
C_s	Pa		Leaf surface CO_2 partial pressure
R_{dark}	µmolC m^{-2} leaf s^{-1}		Dark respiration rate (import Gemini layer data)
g_{fac}	dimensionless	15	Stomatal sensitivity coefficient (does not vary with layer)
Ψ_{leaf}	MPa		Leaf water potential at saturation (does not vary with layer)
E			Leaf transpiration rate
e_a	kPa		actual vapour pressure (see table 1)
e_s	kPa		saturation vapour pressure (see table 1)
LAI	m^{-2}leaf m^{-2}soil		Leaf area index
I_{fac}	dimensionless		Coefficient representing the extent to which R_{dark} is inhibited in the light (import Gemini layer data)
Input Variable			
C_a	Pa		Atmospheric CO_2 partial pressure
h_s	0-1		Leaf surface relative humidity
$PPFD$	µmol m^{-2} s^{-1}		Photosynthetic photon flux density

ANNEXE 4 : Interface de Berkeley Madonna

Figure 1 : Implémentation du modèle de Sinclair sous Berkeley Madonna

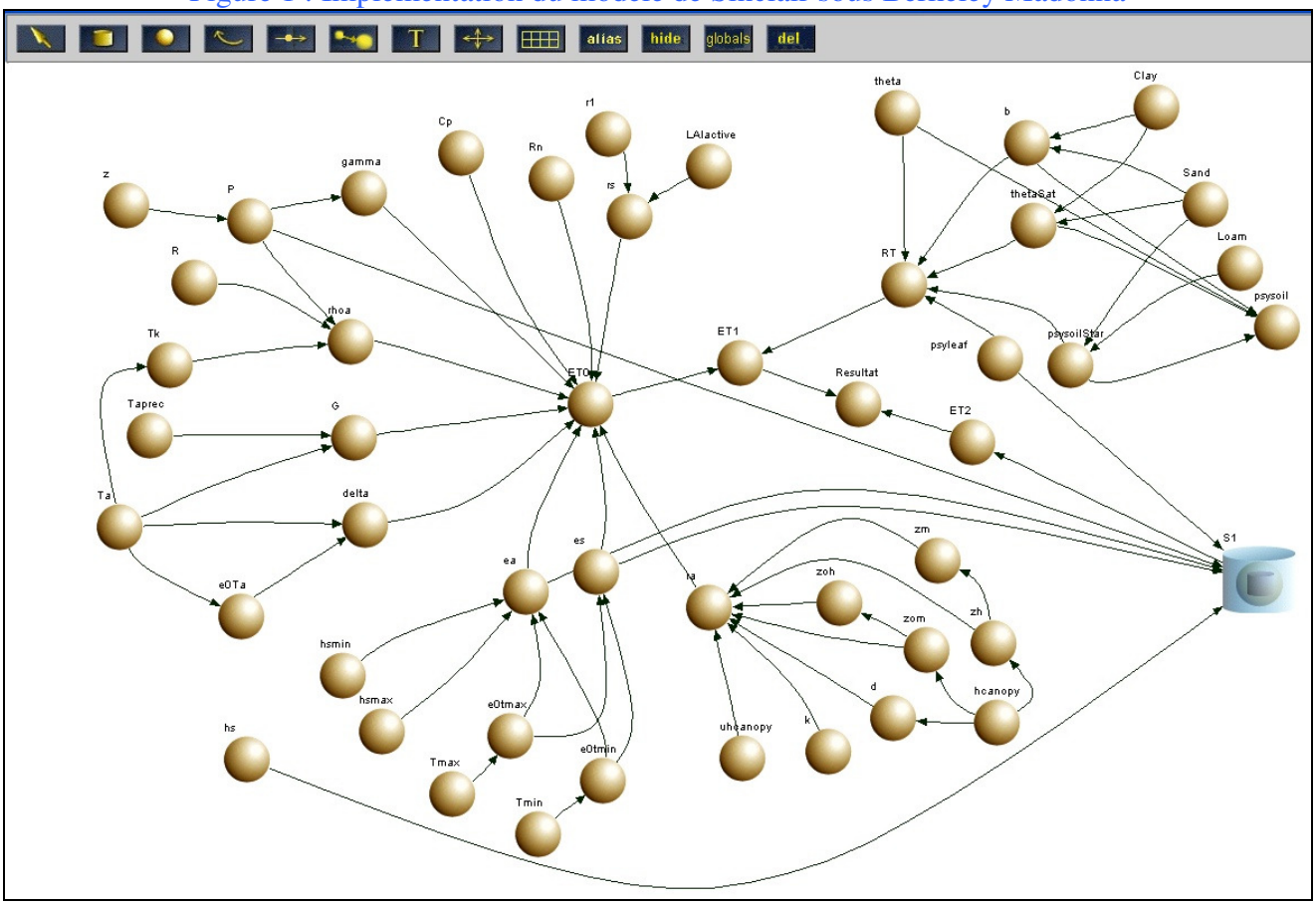
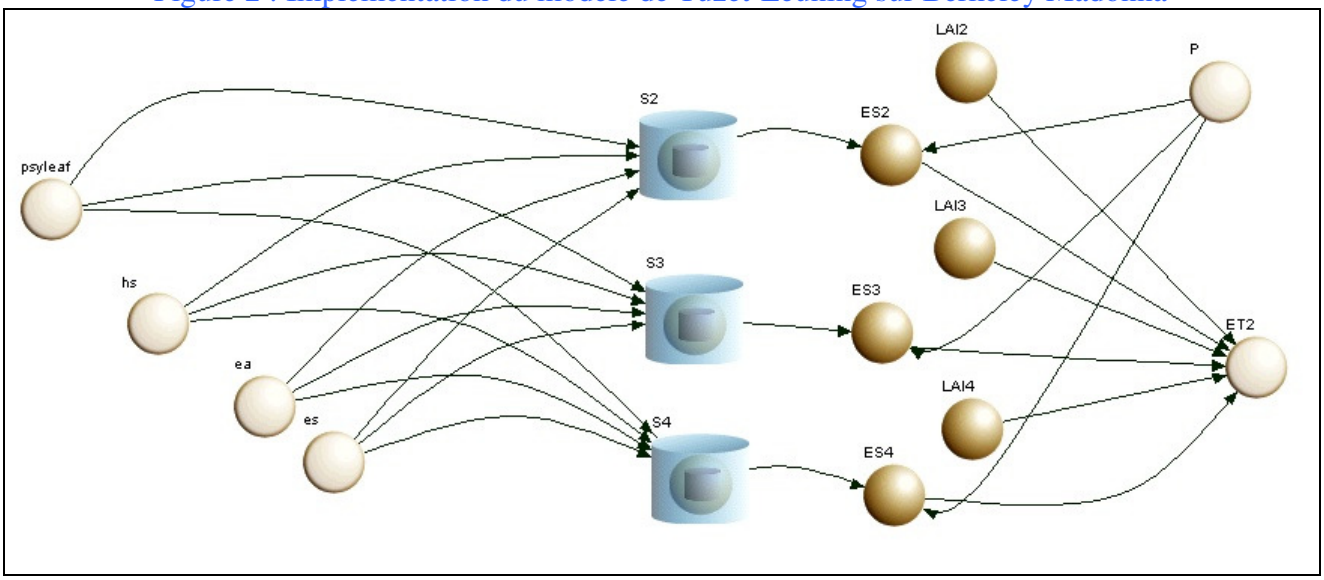


Figure 2 : Implémentation du modèle de Tuzet-Leuning sur Berkeley Madonna



Comme on peut le voir sur les figures 1 et 2, l'implémentation d'un modèle est très simple à réaliser. Les boutons du menu (figure 1) permettent de créer (dans l'ordre) des réservoirs (dérivation/intégration), des entités (paramètre/opération), un lien entre deux entités, un flux, un sous-modèle… Une fois les équations bien posées et organisées, la création des paramètres et leur liaison est évidente.